



DesignWare® ARCV2 ISA

Programmer's Reference Manual for ARC EM Processors

Copyright Notice and Proprietary Information

© 2021 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <https://www.synopsys.com/company/legal/trademarks-brands.html>.

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
www.synopsys.com

Contents

Web Resources	69
Accessing SolvNetPlus	69
Customer Support	69
Accessing ARC Application Notes	70
Synopsys Statement on Inclusivity and Diversity	71
.....	71
Document Revision History	72
 Chapter 1	
Build and Auxiliary Register List	79
1.1 Build Configuration Registers	87
1.1.1 Serializing Auxiliary-Register Writes	89
 Chapter 2	
Exceptions List	91
 Part 1	
ARCV2 Baseline ISA	99
 Chapter 3	
Introduction	101
3.1 Typographic Conventions	101
3.2 Key Features of the ARCV2 ISA	101
3.3 Programmer's Model	103
3.3.1 Core Register Set	103
3.3.2 Auxiliary Register Set	103
3.3.3 32-bit Instruction Formats	103
3.3.4 16-bit Instruction Formats	104
3.3.5 Operating Privileges	104
3.4 Configurability	105
3.4.1 Programming Model Options	105
3.4.2 Processor Component Options	107
3.4.3 ISA Options	108
3.5 Custom Extensions	112
3.5.1 Extension Core Registers	112
3.5.2 Extension Auxiliary Registers	112
3.5.3 Extension Instructions	112
3.5.4 Extension Condition Codes	112
 Chapter 4	

Data Organization and Addressing	115
4.1 The ARCV2 Address Space	115
4.2 Data Formats	116
4.2.1 Vector Operands	116
4.2.2 32-Bit Operand Data Elements	118
4.2.3 Expansion of Literals	118
4.2.4 Expansion of Literals in Vector Instructions	118
4.2.5 Expansion of Literals in Store Double (STD)	119
4.3 Floating-Point Data Formats	120
4.3.1 Single-Precision Floating-Point Data Formats	120
4.3.2 Double-Precision Floating-Point Data Formats	120
4.4 Double Precision Floating-Point Data Formats	121
4.4.1 NaN Formats	121
4.5 Data Layout in Memory	122
4.5.1 64-Bit Data	123
4.5.2 32-Bit Data	124
4.5.3 16-Bit Data	124
4.5.4 8-Bit Data	125
4.5.5 1-Bit Data	125
4.6 Instruction Layout in Memory	126
4.6.1 Addressing Modes	127
4.6.2 Null Instruction Format	128
4.6.3 Conditional Execution	128
4.6.4 Conditional Branch Instruction	129
4.6.5 Compare and Branch Instruction	129
Chapter 5	
Core Architectural State Registers	131
5.1 Core Register Set	131
5.1.1 Hardware Initialization of Register File	133
5.1.2 Core Register Mapping Used in 16-bit Instructions	133
5.1.3 Reduced Configuration of Core Registers	134
5.1.4 Multiple Register Banks	135
5.1.5 Illegal Core Register Usage	137
5.1.6 Pointer Registers, GP (r26), FP (r27), SP (r28)	137
5.1.7 Link Registers, ILINK (r29), BLINK (r31)	137
5.1.8 Long Immediate Data Indicator Register, (r30)	138
5.1.9 Loop Count Register, LP_COUNT (r60)	138
5.1.10 Reserved Register (r61)	139
5.1.11 Immediate Data Indicator, L IMM (r62)	139
5.1.12 Word-aligned Program Counter, PCL (r63)	139
5.2 Extension Core Registers	139
5.2.1 Illegal Extension Core Register Usage	140
5.3 Auxiliary Register Set	141
5.3.1 Baseline Auxiliary Registers	141
5.3.2 Optional Architectural, Exception, and Interrupt State Auxiliary Registers	142
5.3.3 Register Access Permissions	142
5.3.4 Optional Instruction Set Auxiliary Registers	143
5.3.5 User Extension Auxiliary Registers	144

5.3.6	Optional Build Configuration Registers	145
5.3.7	Reset Value of Auxiliary Registers	145
5.3.8	Loop Start Register, LP_START	146
5.3.9	Loop End Register, LP_END	147
5.3.10	Core Identity Register, IDENTITY	148
5.3.11	Program Counter, PC	150
5.3.12	Secure Status Register, SEC_STAT	152
5.3.13	Status Register, STATUS32	155
5.3.14	Status Register Priority 0, STATUS32_P0	163
5.3.15	Saved User Stack Pointer, AUX_USER_SP	164
5.3.16	Interrupt Context Saving Control Register, AUX_IRQ_CTRL	165
5.3.17	Interrupt Vector Base Register, INT_VECTOR_BASE	167
5.3.18	Secure Interrupt Vector Base Register, INT_VECTOR_BASE_S	168
5.3.19	Saved Normal Kernel Stack Pointer, AUX_KERNEL_SP	169
5.3.20	Saved Secure User Stack Pointer, AUX_SEC_U_SP	170
5.3.21	Saved Secure Kernel Stack Pointer, AUX_SEC_K_SP	171
5.3.22	Saved Shadow Normal Stack Pointer, AUX_NSEC_SP	172
5.3.23	Active Interrupts Register, AUX_IRQ_ACT	173
5.3.24	Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING	174
5.3.25	Software Interrupt Trigger, AUX_IRQ_HINT	175
5.3.26	Interrupt Priority Register, IRQ_PRIORITY	176
5.3.27	Secure Jump and Link Indexed Top Address, NSC_TABLE_TOP	178
5.3.28	Secure Jump and Link Indexed Base Address, NSC_TABLE_BASE	179
5.3.29	Jump and Link Indexed Base Address, JLI_BASE	180
5.3.30	Load Indexed Base Address, LDI_BASE	181
5.3.31	Execute Indexed Base Address, EI_BASE	182
5.3.32	Exception Return Address, ERET	183
5.3.33	Exception Return Branch Target Address, ERBTA	184
5.3.34	Exception Return Status, ERSTATUS	185
5.3.35	Exception Cause Register, ECR	186
5.3.36	Exception Fault Address, EFA	188
5.3.37	Exception Secure Status Register, ERSEC_STAT	189
5.3.38	Secure Exception Register, AUX_SEC_EXCEPT	191
5.3.39	Interrupt Cause Registers, ICAUSE	193
5.3.40	Interrupt Select, IRQ_SELECT	194
5.3.41	Interrupt Enable Register, IRQ_ENABLE	195
5.3.42	Interrupt Trigger Register, IRQ_TRIGGER	196
5.3.43	Interrupt Status Register, IRQ_STATUS	197
5.3.44	User Mode Extension Enable Register, XPU	198
5.3.45	Branch Target Address, BTA	200
5.3.46	Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL	202
5.3.47	Interrupt Pending Register, IRQ_PENDING	203
5.3.48	User Extension Flags Register, XFLAGS	204
5.4	Build Configuration Registers	204
5.4.1	Build Configuration Registers Version, BCR_VER	206
5.4.2	BTA Configuration Register, BTA_LINK_BUILD	207
5.4.3	Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD	208
5.4.4	Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD	209
5.4.5	Core Register File Configuration Register, RF_BUILD	210

5.4.6 Multiplier Configuration Register, MULTIPLY_BUILD	212
5.4.7 Swap Instruction Configuration Register, SWAP_BUILD	213
5.4.8 Normalize Instruction Configuration Register, NORM_BUILD	214
5.4.9 Min/Max Instruction Configuration Register, MINMAX_BUILD	215
5.4.10 Barrel Shifter Configuration Register, BARREL_BUILD	216
5.4.11 Instruction Set Configuration Register, ISA_CONFIG	217
5.4.12 Core Architecture Build Configuration Register, ARCH_CONFIG	220
5.4.13 Interrupt Build Configuration Register, IRQ_BUILD	221
Chapter 6	
Interrupts and Exceptions	223
6.1 Introduction	223
6.2 Privileges and Operating Modes	223
6.2.1 Kernel Mode	224
6.2.2 User Mode	224
6.2.3 Privilege Violations	224
6.2.4 Switching Between Operating Modes and Privilege Levels	227
6.2.5 Register Replication	229
6.3 Interrupts	230
6.3.1 Interrupt Unit Features	230
6.3.2 Interrupt Unit Configuration	231
6.3.3 Interrupt Unit Programming	234
6.3.4 Interrupt Handling	239
6.4 Exceptions	262
6.4.1 Exception Precision	262
6.4.2 Exception Vectors and the Exception Cause Register	263
6.4.3 Exception Types and Priorities	266
6.4.4 Exception Detection	287
6.4.5 Effect of Exceptions and Interrupts on Operating Mode	287
6.4.6 Exception Entry	287
6.4.7 Exception Exit	291
6.4.8 Exceptions and Delay Slots	295
6.4.9 Emulation of Extension Instructions	297
6.4.10 Emulation of Extension Registers and Condition Codes	297
Chapter 7	
Instruction Set Summary	299
7.1 Instruction Set Encoding	299
7.1.1 Top-level Instruction Formats	299
7.1.2 Instruction Set Profiles	301
7.1.3 Assignment of Instruction Formats to Major Opcodes	302
7.1.4 32-bit Instruction Formats	304
7.1.5 16-bit Instruction Formats	305
7.1.6 Encoding Notation	306
7.1.7 Long Immediate Source Operands and Null Destination Operands	307
7.1.8 Condition Code Tests	308
7.1.9 Encoding Branch and Jump Delay Slot Modes	309
7.1.10 Encoding Static Branch Predictions	309
7.1.11 Encoding Load / Store Address Write-back Modes	310

7.1.12	Encoding Load / Store Cache Bypass Mode	311
7.1.13	Encoding Load / Store Data Sizes	311
7.1.14	Encoding Load Data Extension Modes	312
7.1.15	Encoding Store Data Source Mode	312
7.1.16	Use of Reserved Encodings	312
7.1.17	Use of Illegal Encodings	313
7.2	Instruction Syntax Conventions	313
7.3	Status flags and conditions	315
7.3.1	Flag Setting	315
7.3.2	Status Flags Notation	315
7.4	Arithmetic and Logical Instructions	316
7.4.1	Integer Adder Operations	316
7.4.2	Move Operations	317
7.4.3	Bit-wise Logical Operations	318
7.4.4	Bit Operations and Mask Operations	319
7.4.5	Shift and Rotate Operations	319
7.4.6	Selection Operations	321
7.4.7	Byte-swapping Operations	321
7.4.8	Bit-scanning Operations	321
7.4.9	Relational Comparison Operations	322
7.4.10	Integer Multiply, Multiply-accumulate, and Divide Operations	322
7.4.11	Dual and Quad Integer Multiply / Accumulate Operations	323
7.4.12	Integer Vector Operations	323
7.5	Memory Access Instructions	324
7.5.1	Load Instructions	325
7.5.2	Store Instructions	326
7.5.3	Stack Pointer Operations	327
7.5.4	Atomic Memory Operations	327
7.5.5	Prefetch Instructions	328
7.6	Auxiliary Register Operations	330
7.6.1	Load from Auxiliary Register	330
7.6.2	Store to Auxiliary Register	330
7.6.3	Auxiliary Register Exchange	330
7.7	Control Flow Instructions	330
7.7.1	Branch Instructions	332
7.7.2	Jump Instructions	335
7.7.3	Zero Overhead Loop Instruction	335
7.7.4	Return from Interrupt or Exception Instruction	335
7.8	Special Instructions	335
7.8.1	Breakpoint Instruction	337
7.8.2	Sleep Instruction	337
7.8.3	Software Interrupt Instruction	337
7.8.4	SETI	337
7.8.5	CLRI	337
7.8.6	Trap Instruction	338
7.8.7	Synchronization Instructions	338
7.9	APEX Extension Instructions	338
7.9.1	Syntax for Generic Extension Instructions	338
7.9.2	Syntax for Single Operand Extension Instructions	339

7.9.3 Syntax for Zero Operand Extension Instructions	339
7.10 Floating-Point Instructions	340
7.11 DSP Instructions	343
7.11.1 Basic Saturating Arithmetic Operations	344
7.11.2 Vector Unpacking Operations	345
7.11.3 Vector ALU Operations	346
7.11.4 Accumulator Operations	348
7.11.5 Vector SIMD 16x16 MAC Operations	348
7.11.6 Dual Inner Product 16x16 MAC Operations	349
7.11.7 32x32 and 16x16 MAC Operations	350
7.11.8 Dual 16x8 MAC Operations	352
7.11.9 32x16 MAC Operations	352
7.11.10 Complex Operations	354
7.11.11 Vector 64-Bit Operations	356
7.11.12 Divide and Square-Root Instructions	356
7.11.13 Address Generation Instructions	357
Chapter 8	
32-bit Instruction Formats Reference	359
8.1 Branch Format [F32_BR0]	360
8.1.1 Branch-Conditional Sub-format [F32_BR0, COND]	360
8.1.2 Branch Unconditional Far Sub-format [F32_BR0, UCOND_FAR]	360
8.2 BRcc, BBITn and BL Format [F32_BR1]	361
8.2.1 BRcc / BBITn Register-register Format [F32_BR1, BCC, REG_REG]	361
8.2.2 Register-immediate Format [F32_BR1, BCC, REG_U6]	362
8.2.3 Branch-and-link Format [F32_BR1, BL]	363
8.3 Load Register with Offset Format [F32_LD_OFFSET]	365
8.4 Store Register with Offset Format [F32_ST_OFFSET]	366
8.5 General Operations Format [F32_GEN4]	367
8.5.1 Operator Format Indicators	367
8.5.2 Operand Format Indicators	367
8.5.3 Syntax and Encoding of Instructions in General Operations Formats	368
8.5.4 Dual-operand Instructions, F32_GEN4	371
8.5.5 Move and Compare Instructions, 0x04, [0x0A – 0x0D] and 0x04, [0x11]	373
8.5.6 Jump and Jump-and-Link Conditionally, 0x04, [0x20 – 0x23]	373
8.5.7 Load Register-Register, 0x04, [0x30 – 0x37]	373
8.5.8 Single Operand Instructions, F32_GEN4	374
8.5.9 Zero Operand Instructions, F32_GEN4	374
8.6 F32_EXT5	375
8.6.1 Dual-operand Instructions, F32_EXT5	375
8.6.2 Single-operand Instructions, F32_EXT5	376
8.6.3 Zero-operand Instructions, F32_EXT5	377
8.7 F32_EXT6	378
8.7.1 Dual-operand Instructions, F32_EXT6	378
8.7.2 Single-operand Instructions, F32_EXT6	378
8.7.3 Zero-operand Instructions, F32_EXT6	379
8.8 APEX Extension Instruction Format [F32_APEX]	380
8.8.1 Extension ALU Operation, Register with Unsigned 6-bit Immediate	380
8.8.2 Extension ALU Operation, Register with Signed 12-bit Immediate	380

8.8.3 Extension ALU Operation, Conditional Register	380
8.8.4 Extension ALU Operation, Conditional Reg with Unsigned 6-bit Immediate	381
8.8.5 FastMath Extension Pack Instructions	381
8.8.6 CryptoPack Instructions	382
Chapter 9	
16-bit Instruction Formats Reference	383
9.1 Compact Move/Load [F16_COMPACT]	385
9.2 Compact Load/Add/Sub [F16_LD_ADD_SUB]	386
9.3 Compact Load/Store [F16_LD_ST_1]	387
9.4 Indexed Jump or Execute [F16_JLI_EI]	388
9.5 Load / Add Register-Register [F16_LD_ADD_RR]	389
9.6 Add/Sub/Shift Register-Immediate [F16_ADD_IMM]	390
9.7 Dual Register Operations [F16_OP_HREG]	391
9.7.1 Long Immediate Operands in Format 0x0E	392
9.8 General Register Format Instructions [F16_GEN_OP]	393
9.8.1 DOP-format General Operations	393
9.8.2 SOP-format 16-bit General Operations	394
9.8.3 ZOP-format 16-bit General Operations	395
9.9 16-bit Load and Store Formats with Offset	397
9.10 Shift/Subtract/Bit Immediate [F16_SH_SUB_BIT]	398
9.11 Stack-based Operations [F16_SP_OPS]	399
9.12 Load/Add GP-Relative [F16_GP_LD_ADD]	400
9.13 Load PCL-Relative [F16_PCL_LD]	401
9.14 Move Immediate [F16_MV_IMM]	402
9.15 ADD/CMP Immediate [F16_OP_IMM]	403
9.16 Branch on Compare Register with Zero [F16_BCC_REG]	404
9.17 Branch Conditionally [F16_BCC]	405
9.17.1 Branch Conditionally with cc Field, 0x1E, [0x03, 0x00 – 0x07]	405
9.18 Branch and Link Unconditionally [F16_BL]	407
Chapter 10	
Instruction Set Details	409
10.1 Instruction List	410
10.2 ARC Instructions	417
ABS	418
ADC	420
ADD	422
ADD1	424
ADD2	426
ADD3	428
AEX	430
AND	432
ASL	434
ASL Multiple	436
ASR	439
ASR multiple	441
ASR16	444
ASR8	446

B	448
B_S	450
BBIT0	452
BBIT1	454
Bcc	456
Bcc_S	458
BCLR	460
BI	462
BIC	464
BIH	466
BLcc	468
BMSK	471
BMSKN	474
BRcc	476
BRK	479
BSET	482
BSPEEK	484
BSPOP	487
BSPUSH	490
BTST	493
BXOR	496
CLRI	498
CMP	500
DADDH11	502
DADDH12	504
DADDH21	506
DADDH22	508
DEXCL1	510
DEXCL2	512
DBNZ	514
DIV	516
DIVU	519
DMULH11	522
DMULH12	524
DMULH21	526
DMULH22	528
DSUBH11	530
DSUBH12	532
DSUBH21	534
DSUBH22	536
EI_S	538
ENTER_S	540
EX	544
EXTB	547
EXTH	549
FCVT32	551
FFS	554
FLAG	556
FLS	561

FSABS	563
FSADD	565
FSCMP	568
FSCMPF	571
FSDIV	574
FSMADD	577
FSMSUB	580
FSMUL	583
FSNEG	586
FSSQRT	588
FSSUB	590
J	593
Jcc	596
JL	599
JLcc	602
JLI_S	605
KFLAG	607
LD LDH LDW LDB	612
LDI, LDI_S	616
LEAVE_S	618
LP<cc>	622
LR	629
LSL16	631
LSL8	633
LSR	635
LSR multiple	637
LSR16	640
LSR8	642
MAX	644
MIN	647
MOV	650
MPY, MPY_S	652
MPYM MPYH	655
MPYMU MPYHU	658
MPYU	661
MPYUW, MPYUW_S	663
MPYW, MPYW_S	665
NEG	667
NOP	669
NEXTTASK	671
NORM	673
NORMH NORMW	675
NOT	678
OR	680
POP	682
POP_S	683
PREFETCH	685
PUSH	688
PUSH_S	689

RCMP	691
REM	693
REMU	696
RLC	699
ROL	701
ROL8	703
ROR	705
ROR multiple	707
ROR8	710
RRC	712
RSUB	714
RTIE	716
SBC	720
SETcc	722
SETI	725
SEXB	728
SEXH SEXW	730
SFLAG	732
SLEEP	734
SJLI	740
SR	742
ST ST STH STB	744
SUB	747
SUB1	750
SUB2	752
SUB3	754
SWAP	756
SWAPE	758
SWI	760
SYNC	763
TRAP_S	766
TST	768
UNIMP_S	770
WEVT	772
XBFU	775
XOR	777

Chapter 11

The Host	779
11.1 The Host Interface	779
11.2 Core and Auxiliary Registers	780
11.3 Memory	780
11.4 Halting	780
11.5 Starting	781
11.6 Single Instruction Stepping	781
11.6.1 SLEEP Instruction in Single Instruction Step Mode	782
11.6.2 BRK Instruction in Single Instruction Step Mode	782
11.7 Software Breakpoints	782

Part 2	
Security Features	783
Chapter 12	
Security Feature Overview	785
Chapter 13	
SecureShield™ Technology	787
13.1 Features of SecureShield™ Technology	787
13.1.1 SecureShield™ Technology Programming Model	787
13.1.2 SecureShield Protection Concepts	788
13.1.3 SecureShield Modes	789
13.1.4 Register Replication	790
13.1.5 Configuring SecureShield	792
Chapter 14	
Secure Pipeline	793
14.1 Secure Build Configuration Register, SEC_BUILD	794
Chapter 15	
Secure Memory Protection Unit	797
15.1 Protection Overview	797
15.2 Configuring an MPU	797
15.2.1 Enabling the Memory-Protection Unit	798
15.3 Programming MPU Protection	799
15.3.1 Programming Considerations	799
15.4 Software Lookup of the MPU Table	800
15.5 Protection Mechanism	800
15.5.1 Protection Exception Priority	801
15.5.2 Switching Between Operating Modes	801
15.6 MPU Auxiliary Register Interface	802
15.6.1 MPU Enable Register, MPU_EN	804
15.6.2 MPU Probe Register, MPU_PROBE	811
15.6.3 MPU Build Configuration Register, MPU_BUILD	812
Chapter 16	
Secure Debug Access	815
16.1 Secure Debug Access Overview	815
16.2 Unlocking a Debug Port	815
16.3 Secure Debug Unlock Module Auxiliary Register Interface	816
16.3.1 Key Data Register, AUX_KEY_DATA	817
16.3.2 Key Unlock Register, AUX_KEY_UNLOCK	818
Chapter 17	
Error Protection	819
17.1 Auxiliary Registers	819
17.1.1 Error Protection Hardware Control Register, ERP_CTRL	820
17.1.2 Register File Error Protection Status Register, RFERP_STATUS_0	824
17.1.3 Register File Error Protection Status Register, RFERP_STATUS_1	826
17.1.4 ECC Syndrome Register, ESYN	827
17.1.5 Error Protection Configuration Register, ERP_BUILD	828

17.2 Exceptions	830
Part 3	
Memory and System Components	831
Chapter 18	
Memory Protection Unit	833
18.1 Memory Protection Unit	833
18.1.1 Key Features	834
18.1.2 Configuration Options	834
18.1.3 Enabling the MPU	834
18.1.4 Data Organization and Addressing	834
18.1.5 Modes of Operation	840
18.1.6 Memory Protection Unit Registers	843
18.1.7 MPU Enable Register, MPU_EN	845
18.1.8 MPU Exception Cause Register, MPU_ECR	848
18.1.9 MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB15	850
18.1.10 MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP15	851
18.1.11 Memory Protection Build Configuration Register, MPU_BUILD	854
18.1.12 MPU Exceptions	855
Chapter 19	
Protection Schemes	857
19.1 Stack Checking	857
19.1.1 Build Configuration Register	858
19.1.2 Enabling Stack Checking	858
19.1.3 Specifying Stack Regions	859
19.1.4 Stack-Checking Operation	860
19.1.5 Exception Information	863
19.1.6 Stack Protection Out-of-Bound Limitations	863
19.1.7 Stack Region Auxiliary Register Set	863
19.1.8 Stack Checking Exceptions	870
19.2 Coexisting Protection Schemes	871
19.3 Code Protection	872
19.3.1 Code Protection Scheme	872
19.3.2 Code Protection Behavior	872
19.3.3 Code Protection Register, CPROT_BUILD	874
Chapter 20	
ICCM	875
20.1 ICCM Auxiliary Registers	875
20.1.1 ICCM base address, AUX_ICCM	876
20.1.2 ICCM Partition Size Register, CCM_PART_SIZE	877
20.1.3 ICCM Configuration Register, ICCM_BUILD	878
20.2 Exceptions	879
Chapter 21	
Instruction Fetch Queue	881
21.1 IFQ Introduction	881
21.2 Invalidate Instruction Cache, IC_IVIC	882

21.2.1 Instruction Fetch Queue Configuration Register, IFQUEUE_BUILD	883
Chapter 22	
Instruction Cache	885
22.1 Instruction Cache Auxiliary Registers	885
22.2 Auxiliary Registers	885
22.2.1 Invalidate Instruction Cache, IC_IVIC	887
22.2.2 Instruction Cache Control Register, IC_CTRL	888
22.2.3 Lock Instruction Cache Line, IC_LIL	890
22.2.4 Invalidate Instruction-Cache Start Region, IC_IVIR	892
22.2.5 Invalidate Instruction-Cache End Region, IC_ENDR	893
22.2.6 Invalidate Instruction-Cache Line, IC_IVIL	894
22.2.7 Instruction Cache External-Access Address, IC_RAM_ADDR	896
22.2.8 Instruction-Cache Tag Access, IC_TAG	900
22.2.9 Instruction Cache Secure Bit Tag Register, IC_XTAG	904
22.2.10 Instruction Cache Data Access, IC_DATA	905
22.2.11 Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG	908
22.2.12 Instruction Cache Configuration Register, I_CACHE_BUILD	909
22.3 Exceptions	910
Chapter 23	
DCCM	911
23.1 DCCM Auxiliary Registers	911
23.1.1 DCCM Base Address, AUX_DCCM	912
23.1.2 DCCM Configuration Register, DCCM_BUILD	913
23.1.3 Exceptions	914
Chapter 24	
Data Cache	915
24.1 Data Cache Auxiliary Registers	915
24.1.1 Invalidate Data Cache, DC_IVDC	917
24.1.2 Data Cache Control Register, DC_CTRL	919
24.1.3 Lock Data Cache Line, DC_LDL	921
24.1.4 Invalidate Data Line, DC_IVDL	924
24.1.5 Flush Data Cache, DC_FLSH	926
24.1.6 Flush Data Line, DC_FLDL	928
24.1.7 Data-Cache Region Start Address, DC_STARTR	930
24.1.8 Data-Cache Region End Address, DC_ENDR	931
24.1.9 Data Cache External Access Address, DC_RAM_ADDR	932
24.1.10 Data Cache Tag Access, DC_TAG	934
24.1.11 Data Cache Secure Bit Register, DC_XTAG	936
24.1.12 Data Cache Data Access, DC_DATA	937
24.1.13 Non-cached Memory Region, AUX_CACHE_LIMIT	938
24.2 Build Configuration Registers	938
24.2.1 Data Cache Configuration Register, D_CACHE_BUILD	940
24.2.2 Exceptions	941
24.3 Cache Maintenance	941
24.3.1 Cache-based Purge and Flush Operations	941
24.3.2 Region-based Flush	942
24.3.3 Region-Based Purge	942

Chapter 25	
Task Switching Component	943
25.1 Overview	943
25.1.1 Address Translation	943
25.1.2 JLAT Segment 1 Base Address, JOB_BASE1	945
25.1.3 JLAT Segment 2 Base Address, JOB_BASE2	946
25.1.4 Task Switch Build Register, TASK_SWITCH_BUILD	947
Chapter 26	
Overlay Management Unit	949
26.1 Overview	949
26.1.1 Auxiliary Registers for Overlay Manager Unit	949
26.1.2 Start of Translation Region, OLM_VA_START	950
26.1.3 End of Translation Region, OLM_VA_END	951
26.1.4 TLB Page Descriptor Register for 32-Bit Virtual Address, TLBPD0	952
26.1.5 TLB Page Descriptor Register for 32-Bit Physical Address, TLBPD1	954
26.1.6 TLB Index Register, TLBIndex	957
26.1.7 TLB Command Register, TLBCommand	959
26.1.8 TLBWrite and TLBWriteNI	960
26.1.9 TLBRead	961
26.1.10 TLBGetIndex	962
26.1.11 TLBProbe	962
26.1.12 IVUTLB	963
26.1.13 TLBInsertEntry, TLBInsertEntryJ Commands	964
26.1.14 TLBDeleteEntry	964
26.1.15 TLBResetLRU	965
26.1.16 TLB Enable Register, TLBEnable	966
26.1.17 Scratch Data Register, SCRATCH_DATA0	967
26.2 OLM Build Configuration Register, OLM_BUILD	968
Chapter 27	
Program Storage	971
27.1 NV_ICCM Register Interface	971
27.1.1 NV_ICCM Control Register, NV_ICCM_CTRL	972
27.1.2 NV_ICCM Erase Register, NV_ICCM_ERASE	974
27.1.3 NV_ICCM Programming Register, NV_ICCM_PROG	975
27.1.4 NV_ICCM Data Register, NV_ICCM_DATA	976
27.2 Exceptions	976
Chapter 28	
Cryptographic Key Storage	977
28.1 Cryptographic Key Storage Introduction	977
28.2 Cryptographic Key Storage Register Interface	977
28.2.1 Key Store Control Register, KEY_STORE_CTRL	979
28.2.2 Key Store Password Register, KEY_STORE_PWD	981
28.2.3 Key Store Address Register, KEY_STORE_ADDR	982
28.2.4 Key Store Data Register, KEY_STORE_DATA	983
28.2.5 Key Store SID Register, KEY_STORE_SID	984
28.2.6 Key Store Build Configuration Register, KEY_STORE_BUILD	985

Chapter 29

Calibration Parameter Storage	987
29.0.1 Calibration Parameter Store Control Register, CAL_STORE_CTRL	988
29.0.2 Calibration Parameter Store Address Register, CAL_STORE_ADDR	990
29.0.3 Calibration Parameter Store Data Register, CAL_STORE_DATA	991
29.1 Build Configuration Registers	991
29.1.1 Calibration Parameter Store Build Configuration Register, CAL_STORE_BUILD	992

Chapter 30

Processor Timers	993
30.1 Timers	993
30.2 Auxiliary Timer Registers	994
30.2.1 Timer 0 Count Register, COUNT0	996
30.2.2 Timer 0 Control Register, CONTROL0	997
30.2.3 Timer 0 Limit Register, LIMIT0	999
30.2.4 Timer 1 Count Register, COUNT1	1000
30.2.5 Timer 1 Control Register, CONTROL1	1001
30.2.6 Timer 1 Limit Register, LIMIT1	1002
30.2.7 Secure Timer 0 Count Register, AUX_ST0_COUNT	1003
30.2.8 Secure Timer 0 Control Register, AUX_ST0_CTRL	1004
30.2.9 Secure Timer 0 Limit Register, AUX_ST0_LIMIT	1005
30.2.10 Secure Timer 1 Count Register, AUX_ST1_COUNT	1006
30.2.11 Secure Timer 1 Control Register, AUX_ST1_CTRL	1007
30.2.12 Secure Timer 1 Limit Register, AUX_ST1_LIMIT	1008
30.2.13 RTC Control Register, AUX_RTC_CTRL	1009
30.2.14 RTC Count Low Register, AUX_RTC_LOW	1011
30.2.15 RTC Count High Register, AUX_RTC_HIGH	1012
30.2.16 Timers Configuration Register, TIMER_BUILD	1013

Chapter 31

Enhanced Watchdog Timer	1015
31.1 Watchdog Timer Index	1015
31.2 Registers	1016
31.2.1 Watchdog Password Register, EWDT_PASSWD	1017
31.2.2 Watchdog Time Error Status Register, EWDT_ESR	1018
31.2.3 Watchdog Timer Index Register, EWDT_INDEX	1019
31.2.4 Watchdog Control Register, EWDT_CTRLx	1020
31.2.5 Watchdog Timer Period Register, EWDT_PERIODx	1022
31.2.6 Watchdog Timer Counter Register, EWDT_COUNTx	1023
31.2.7 Watchdog Timer Build Configuration Register, EWDT_BUILD	1024
31.3 Programming the WD Timer	1025
31.3.1 Countdown Timer Mode	1025

Chapter 32

Peripheral Bus	1027
32.1 DMP Auxiliary Registers	1027
32.1.1 Peripheral Memory Region, DMP_PER_AUX	1028
32.1.2 Peripheral Region Build Configuration Register, DMP_PP_BUILD	1030

Chapter 33

External Host Debugging	1031
33.1 External Host Debugging Introduction	1031
33.2 Host Debug Register Interface	1031
33.2.1 Debug Register, DEBUG	1032
33.2.2 Interrupt and Register Bank Debug Register, DEBUGI	1036
33.2.3 Architectural Clock Gating Control Register, ACG_CTRL	1038
Chapter 34	
Debug Features	1039
34.1 Debug Introduction	1039
34.2 Actionpoints	1039
34.2.1 Actionpoint Auxiliary Registers	1039
34.2.2 Actionpoint Extensions to Debug Register, DEBUG, 0x05	1043
34.2.3 Action Points and Architectural Clock Gating	1044
34.2.4 Actionpoints Configuration Register, AP_BUILD	1044
34.2.5 Actionpoint Match Value, AP_AMV0	1045
34.2.6 Actionpoint Match Mask, AP_AMM0	1046
34.2.7 Actionpoint Control, AP_AC0	1047
34.2.8 Actionpoint Match Value, AP_AMV1	1052
34.2.9 Actionpoint Match Mask, AP_AMM1	1053
34.2.10 Actionpoint Control, AP_AC1	1054
34.2.11 Actionpoint Match Value, AP_AMV2	1055
34.2.12 Actionpoint Match Mask, AP_AMM2	1056
34.2.13 Actionpoint Control, AP_AC2	1057
34.2.14 Actionpoint Match Value, AP_AMV3	1058
34.2.15 Actionpoint Match Mask, AP_AMM3	1059
34.2.16 Actionpoint Control, AP_AC3	1060
34.2.17 Actionpoint Match Value, AP_AMV4	1061
34.2.18 Actionpoint Match Mask, AP_AMM4	1062
34.2.19 Actionpoint Control, AP_AC4	1063
34.2.20 Actionpoint Match Value, AP_AMV5	1064
34.2.21 Actionpoint Match Mask, AP_AMM5	1065
34.2.22 Actionpoint Control, AP_AC5	1066
34.2.23 Actionpoint Match Value, AP_AMV6	1067
34.2.24 Actionpoint Match Mask, AP_AMM6	1068
34.2.25 Actionpoint Control, AP_AC6	1069
34.2.26 Actionpoint Match Value, AP_AMV7	1070
34.2.27 Actionpoint Match Mask, AP_AMM7	1071
34.2.28 Actionpoint Control, AP_AC7	1072
34.2.29 Actionpoint Isolation Enable Register, AP_ISO_EN	1073
34.2.30 Watchpoint Program Counter, AP_WP_PC	1074
34.3 SmaRT	1074
34.3.1 Features	1074
34.3.2 Overview of SmaRT Operation	1075
34.3.3 Auxiliary Registers	1076
34.3.4 SMART_BUILD Configuration Register, SMART_BUILD	1077
34.3.5 SmaRT Control Register, SMART_CONTROL	1078
34.3.6 SmaRT Data Register, SMART_DATA	1080
34.3.7 SRC_ADDR Value, Index 0	1080

34.3.8	DEST_ADDR Value, Index 1	1081
34.3.9	FLAGS Value, Index 2	1081
34.4	Debug Interface	1082
34.4.1	ARC Debug-Access Registers	1082
34.4.2	Debug Register, DB_STATUS	1084
34.4.3	Debug Register, DB_CMD	1085
34.4.4	Debug Register, DB_ADDR	1086
34.4.5	Debug Register, DB_DATA	1087
34.4.6	Debug Register, DB_RESET	1088
34.4.7	Initiating a Basic Debug Operation	1088
Chapter 35		
Performance Counters		1091
35.1	Performance Counters Introduction	1091
35.1.1	Performance Counters and Architectural Clock Gating	1092
35.2	Performance Counters Registers	1092
35.2.1	Countable Conditions Index Register, CC_INDEX	1094
35.2.2	Countable Conditions Name0 Register, CC_NAME0	1095
35.2.3	Countable Conditions Name1 Register, CC_NAME1	1096
35.2.4	Count-Value Registers, PCT_COUNTL	1097
35.2.5	Count-Value Registers, PCT_COUNTH	1098
35.2.6	Snapshot-Value Registers, PCT_SNAPL	1099
35.2.7	Snapshot-Value Registers, PCT_SNAPH	1100
35.2.8	Configuration Register, PCT_CONFIG	1101
35.2.9	Control Register, PCT_CONTROL	1103
35.2.10	Index-Select Register, PCT_INDEX	1105
35.2.11	Minimum Value Registers, PCT_MINMAXL	1106
35.2.12	Maximum Value Registers, PCT_MINMAXH	1107
35.2.13	Address-Range Registers, PCT_RANGEL	1108
35.2.14	Address-Range Registers, PCT_RANGEH	1109
35.2.15	User-Flag Register, PCT_UFLAGS	1110
35.2.16	Performance Counter Build-Configuration Register, PCT_BUILD	1112
35.2.17	Countable Conditions Build Configuration Register, CC_BUILD	1113
Chapter 36		
ARC Trace		1115
36.1	ARC Trace Introduction	1115
36.2	ARC Trace Auxiliary Registers	1115
36.2.1	ARC Trace Address Register, RTT_ADDRESS	1116
36.2.2	ARC Trace DATA Register, RTT_DATA	1117
36.2.3	ARC Trace CMD Register, RTT_COMMAND	1118
36.2.4	ARC Trace Build Configuration Register, RTT_BUILD	1119
36.3	ARC Trace Address Map	1120
36.4	ARC Trace Control Registers	1120
36.4.1	ARC Trace I/F Build Configuration Register, ARCT_BUILD	1122
36.4.2	Trace Producer Select Register, ARCT_PRSEL	1124
36.4.3	Trace Timestamp Enable Register, ARCT_TSTAMP_ENABLE	1125
36.4.4	Trace Debugger Message Enable Register, ARCT_DEBUGGER_MESSAGE_ENABLE	1126
36.4.5	ARC Trace Producer Build Configuration Register, RTT_PRDCR_BLD_REG	1127

36.5	ARC Trace Configuration Registers	1127
36.5.1	Trace On-Chip Memory Base Address Register, ARCT_OCM_BASE	1129
36.5.2	Trace On-Chip Memory Size Register, ARCT_OCM_SIZE	1130
36.5.3	Trace On-Chip Memory Write Pointer Register, ARCT_OCM_WPTR	1131
36.5.4	Trace Transport Status Register, ARCT_TR_STATUS	1132
36.5.5	Trace Nexus Offload Control Register, ARCT_FLUSH_CTRL	1134
36.5.6	Trace Nexus Offload Control Register, ARCT_OFFLOAD_CTRL	1135
36.5.7	Pattern Generation Register, PTRN_GEN	1136
36.5.8	Nexus Clock Control Register, NEXUS_CLK_DIV	1137
36.5.9	EVTI Control Register, PR_EVTI_REG	1138
36.6	ARC Trace Producer Registers	1138
36.6.1	Register Description	1139
36.6.2	Address Filter Registers	1140
36.6.3	Memory Write Filter Start Register, MEM_WR_FILTER_START	1144
36.6.4	Memory Write Filter Stop Register, MEM_WR_FILTER_STOP	1145
36.6.5	Memory Read Filter Start Register, MEM_RD_FILTER_START	1146
36.6.6	Memory Read Filter Stop Register, MEM_RD_FILTER_STOP	1147
36.6.7	Data Filter0 LSW Register, PRDATAFILTER0_LSW	1148
36.6.8	Data Filter0 MSW Register, PRDATAFILTER0_MSW	1149
36.6.9	Data Filter1 LSW Register, PRDATAFILTER1_LSW	1150
36.6.10	Data Filter1 MSW Filter, PRDATAFILTER1_MSW	1151
36.6.11	Trace Producer Type Register, ARCT_PRTYPE	1152
36.6.12	Trace Filter Type Register, ARCT_FLT_TYPE	1153
36.6.13	Trace Filter Control Register, ARCT_FLT_CTRL	1154
36.6.14	Trace Producer Source Enable Register, ARCT_PRSRC_E	1156
36.6.15	Trace Producer Watchpoint Status Register, ARCT_PRWP_STATUS	1157
36.6.16	Trace Producer Watchpoint Enable Register, ARCT_PRWP_E	1159
36.6.17	Producer Cross Trigger Enable Register, CTIEN_REG_ADDR	1161
36.6.18	Producer ATB SYNC Frame Insertion Register, SYNCRFR_REG_ADDR	1162
36.6.19	Producer Trace Attribute Register, ARCT_PRATTR	1163
36.6.20	Producer CoreSight ID Register, ATID_REG_ADDR	1165
Chapter 37		
	Power Management Features	1167
37.1	Introduction to Power Management Features	1167
37.2	PDM and DVFS Register Interface	1167
37.2.1	Core Power Status Register, PDM_PSTAT	1168
37.2.2	ARC Trace Power Status Register, RTT_PDM_PSTAT	1170
37.2.3	Core Performance Register, DVFS_PL	1171
37.2.4	Power Down Register, PDM_PMODE	1172
37.3	Build Configuration Registers	1173
37.3.1	Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD	1174
Chapter 38		
	Floating-Point Unit	1175
38.1	Core Register Set	1175
38.2	Extension Core Registers	1176
38.3	FPU Auxiliary Registers	1177
38.3.1	Floating-Point Unit Control Register, FPU_CTRL	1178

38.3.2 Floating-Point Unit Status Register, FPU_STATUS	1180
38.3.3 Double-precision Floating Point D1 Lower Register, AUX_DFP1L	1184
38.3.4 Double-precision Floating Point D1 Higher Register, AUX_DFP1H	1185
38.3.5 Double-precision Floating Point D2 Lower Register, AUX_DFP2L	1186
38.3.6 Double-precision Floating Point D2 Higher Register, AUX_DFP2H	1187
38.4 Build Configuration Registers	1187
38.4.1 Floating-Point Unit Build Register, FPU_BUILD	1188
Chapter 39	
XY	1191
39.1 XY Data Organization	1192
39.1.1 Address Generation	1192
39.1.2 Data Transformation	1194
39.2 XY Extension Core Registers	1195
39.3 XY Register Set	1196
39.3.1 XCCM Base Address, XCCM_BASE	1197
39.3.2 YCCM Base Address, YCCM_BASE	1198
39.3.3 XY Build Configuration Register, XY_BUILD	1199
Chapter 40	
Address Generation Unit Auxiliary Interface	1201
40.1 AGU Auxiliary Register Set	1201
40.1.1 AGU Address Pointer Registers, AGU_AUX_APx	1202
40.1.2 AGU Offset Registers, AGU_AUX_OSx	1207
40.1.3 AGU Modifier Registers, AGU_AUX_MODx	1208
40.1.4 AGU Build Configuration Register, AGU_BUILD	1214
Chapter 41	
Bitstream Processing	1217
41.0.1 Bitstream Control Register, BS_AUX_CTRL	1218
41.0.2 Bitstream Base Address Register, BS_AUX_ADDR	1219
41.0.3 Bitstream Offset Register, BS_AUX_BIT_OS	1220
41.0.4 Bitstream Write Data Register, BS_AUX_WDATA	1221
41.1 Build Configuration Registers	1221
41.1.1 Bitstream Identity Register, BS_BUILD	1222
41.2 Exceptions	1222
Chapter 42	
User Auxiliary Register Interface	1223
42.1 UAUX in Secure Mode	1223
Chapter 43	
ARConnect	1225
43.1 ARConnect Introduction	1225
43.2 ARConnect Register Interface	1225
43.2.1 ARConnect Command Register, CONNECT_CMD	1227
43.2.2 ARConnect Write Data Register, CONNECT_WDATA	1228
43.2.3 ARConnect Read Data Register, CONNECT_READBACK	1229
43.2.4 ARConnect Build Configuration Register, CONNECT_SYSTEM_BUILD	1230
43.2.5 Inter-core Semaphore Unit BCR, CONNECT_SEMA_BUILD	1232

43.2.6	Inter-core Message Unit BCR, CONNECT_MESSAGE_BUILD	1233
43.2.7	Power Management Unit BCR, CONNECT_PMU_BUILD	1235
43.2.8	ARConnect Global Free Running Counter BCR, CONNECT_GFRC_BUILD	1236
43.2.9	Inter-Core Interrupt Vector Control Unit BCR, CONNECT_IVC_BUILD	1237
43.2.10	Inter-Core Interrupt Unit BCR, CONNECT_ICI_BUILD	1238
43.2.11	Inter-Core Debug Unit BCR, CONNECT_ICD_BUILD	1239
43.2.12	ARConnect PDM BCR, CONNECT_PDM_BUILD	1240
43.3	Programming Restrictions	1240
43.3.1	Atomic Operation	1240
43.3.2	Preventing Conflicts	1240
 Chapter 44		
micro DMA Controller		1243
44.1	DMAC Features	1243
44.2	DMAC Auxiliary Register Interface	1243
44.2.1	XY DMA Controller	1245
44.2.2	DMA Controller Configuration Register, DMACTRL	1247
44.2.3	DMA Channel Enable Register, DMACENB	1248
44.2.4	DMA Channel Disable Register, DMACDSB	1249
44.2.5	DMA Channel High Priority Level Register, DMACHPRI	1250
44.2.6	DMA Channel Normal Priority Level Register, DMACNPRI	1251
44.2.7	DMA Channel Transfer Request Register, DMACREQ	1252
44.2.8	DMA Channel Status Register, DMACSTAT0	1254
44.2.9	DMA Channel Status Register, DMACSTAT1	1255
44.2.10	DMA Channel Interrupt Request Status Register, DMACIRQ	1257
44.2.11	DMA Channel Structure Base Address Register, DMACBASE	1258
44.2.12	DMA Channel Control Register, DMACTRLx	1261
44.2.13	DMA Channel Source Address Register, DMASARx	1267
44.2.14	DMA Channel Destination Address Register, DMADARx	1268
44.2.15	DMA Channel Linked-List Pointer Register, DMALLPx	1270
44.2.16	DMA Channel Reset Register, DMACRST	1273
44.2.17	DMA Peripheral Flow Controller Enable Register, DMAFENB	1274
44.2.18	DMA Peripheral Flow Controller Disable Register, DMAFDSB	1275
44.2.19	DMA Build Configuration Register, DMAC_BUILD	1276
 Part 4		
ARCV2 Scalar DSP Extensions		1279
 Chapter 45		
Scalar DSP Introduction		1281
45.1	Key Features of the ARCV2 ISA DSP Extensions	1281
45.2	Programmer's Model	1282
45.3	Configurability	1282
 Chapter 46		
Data Organization and Addressing		1283
46.1	DSP Data Formats	1283
46.1.1	8-Bit Signed Integer Vector	1283
46.1.2	16-Bit Signed Integer	1283
46.1.3	16-Bit Unsigned Integer	1283

46.1.4	16-Bit Signed Fractional	1284
46.1.5	16-Bit Signed Integer Vector	1284
46.1.6	16-Bit Unsigned Integer Vector	1284
46.1.7	16-Bit Signed Fractional Vector	1284
46.1.8	Complex 16+16 Bit Signed Fractional Data	1285
46.1.9	32-Bit Signed Integer	1285
46.1.10	32-Bit Unsigned Integer	1285
46.1.11	32-Bit Signed Fractional	1285
46.1.12	Accumulators	1286
46.1.13	Accumulator Saturation	1288
46.1.14	Accumulator Operations	1288
 Chapter 47		
DSP Auxiliary Interface		1291
47.1	DSP Auxiliary Registers	1291
47.1.1	Accumulator Low Register, ACC0_LO	1292
47.1.2	Low Accumulator Guard and Status Register, ACC0_GLO	1293
47.1.3	Accumulator High Register, ACC0_HI	1294
47.1.4	High Accumulator Guard and Status Register, ACC0_GHI	1295
47.1.5	DSP Butterfly Instructions Data Register, DSP_BFLY0	1296
47.1.6	DSP FFT Control Register, DSP_FFT_CTRL	1297
47.1.7	DSP Control Register, DSP_CTRL	1298
47.2	Build Configuration Registers	1299
47.2.1	DSP Build Configuration Register, DSP_BUILD	1300
 Chapter 48		
Scalar DSP Instruction Set Details		1301
48.1	Notations Used in the DSP Instructions	1301
48.2	List of Instructions	1301
	ABSS	1308
	ABSSH	1310
	ADCS	1312
	ADDS	1314
	ASLACC	1316
	ASLS	1320
	ASLSACC	1322
	ASRS	1327
	ASRSR	1329
	CBFLYHF0R	1331
	CBFLYHF1R	1335
	CMACCHFR	1339
	CMACCHNFR	1343
	CMACHFR	1346
	CMACHNFR	1350
	CMPYCHFR	1353
	CMPYCHNFR	1357
	CMPYHFR	1360
	CMPYHFMR	1364
	CMPYHNFR	1368

DIV	1371
DIVF	1374
DIVU	1377
DMACH	1380
DMACHBL	1383
DMACHBM	1386
DMACHF	1389
DMACHFR	1392
DMACHU	1395
DMPYH	1398
DMPYHBL	1401
DMPYHBM	1403
DMPYHF	1405
DMPYHFR	1408
DMPYHU	1411
DMPYHWF	1413
FLAGACC	1416
GETACC	1418
MAC	1423
MACD	1426
MACDF	1429
MACDU	1432
MACF	1435
MACFR	1438
MACU	1441
MACWHFM	1444
MACWHFMR	1447
MACWHL	1450
MACWHFL	1453
MACWHFLR	1456
MACWHKL	1459
MACWHKUL	1462
MACWHUL	1465
MODIF	1468
MODAPP	1472
MPY	1474
MPY_S	1477
MPYD	1479
MPYDF	1482
MPYDU	1485
MPYF	1488
MPYFR	1491
MPYM MPYH	1494
MPYMU MPYHU	1496
MPYU	1498
MPYUW	1500
MPYUW_S	1502
MPYW	1504
MPYW_S	1506

MPYWHFL	1508
MPYWHFLR	1511
MPYWHFM	1514
MPYWHFMR	1517
MPYWHL	1520
MPYWHUL	1523
MPYWHKL	1526
MPYWHKUL	1529
MSUBDF	1532
MSUBF	1535
MSUBFR	1538
MSUBWHFL	1541
MSUBWHFLR	1544
MSUBWHFM	1547
MSUBWHFMR	1550
NEGS	1553
NEGSH	1555
NORMACC	1557
REM	1561
REMU	1565
RNDH	1569
SATH	1571
SATF	1573
SBCS	1575
SETACC	1577
SQRT	1582
SQRTF	1584
SUBS	1586
VABS2H	1588
VABSS2H	1590
VADD2H	1592
VADD4B	1594
VADDSUB2H	1597
VADDS2H	1599
VADDSUB2H	1601
VADDSUBS2H	1603
VALGN2H	1606
VASL2H	1608
VASLS2H	1610
VASR2H	1613
VASRS2H	1615
VASRSR2H	1618
VEXT2BHL	1621
VEXT2BHLF	1623
VEXT2BHM	1625
VEXT2BHMF	1627
VLSR2H	1629
VMAC2H	1631
VMAC2HF	1633

VMAC2HFR	1637
VMAC2HNFR	1641
VMAC2HU	1644
VMAX2H	1646
VMIN2H	1648
VMPY2H	1650
VMPY2HF	1652
VMPY2HFR	1656
VMPY2HU	1660
VMPY2HWF	1663
VMSUB2HF	1666
VMSUB2HFR	1670
VMSUB2HNFR	1674
VNEG2H	1677
VNEGS2H	1679
VNORM2H	1681
VPACK2HL	1683
VPACK2HM	1685
VPACK2HBL	1687
VPACK2HBLF	1689
VPACK2HBM	1691
VPACK2HBMF	1693
VPERM	1695
VREP2HL	1698
VREP2HM	1700
VSEXT2BHL	1702
VSEXT2BHM	1704
VSUB2H	1706
VSUBADD2H	1708
VSUB2H	1710
VSUBS2H	1712
VSUB4B	1714
VSUBADDS2H	1716

Part 5

Appendixes 1719

Appendix A

ISA Version Differences	1721

Appendix B

Implementation-dependent Behavior	1729
B.1 EM Exception Handling	1729
B.1.1 ECR User-mode Setting in the ARC EM Family of Cores	1729
B.2 Cause Codes for Memory Accesses and Data/ Address Errors	1729
B.3 Exception Handling	1730
B.3.1 ECR User-mode Setting for the ARC EM/EV/VPX5 Family of Cores	1730
B.4 Result on Overflow from Integer Division	1730

Appendix C

Auxiliary Functions for DSP and XY Extensions	1731
C.1 Auxiliary Function that implements the saturation and rounding functionality	1731
C.2 Data Transformation	1732
C.2.1 Realigns Data from Even and Odd XY Banks	1732
C.2.2 Data Mask: Masks Unused MSB Bytes	1732
C.2.3 Data to Container Expansion Function	1732
C.2.4 ReplicateReverse Function	1733
C.2.5 Destination Transformation	1734
C.2.6 Destination Memory Function	1735
C.3 Container to Data Packing	1736
C.3.1 Destination Transformation for Endian Parameters	1737
C.3.2 Destination Transformation for Endian Parameters	1738
Appendix D	
Examples	1739
D.1 Operating Mode Switch	1739
Appendix E	
ARConnect Commands Details	1741
E.1 CMD_CHECK_CORE_ID	1741
E.2 CMD_INTRPT_GENERATE_IRQ	1742
E.3 CMD_INTRPT_GENERATE_ACK	1743
E.4 CMD_INTRPT_READ_STATUS	1744
E.5 CMD_INTRPT_CHECK_SOURCE	1745
E.6 CMD_INTRPT_GENERATE_ACK_BIT_MASK	1746
E.7 CMD_SEMA_CLAIM_AND_READ	1747
E.8 CMD_SEMA_RELEASE	1748
E.9 CMD_SEMA_FORCE_RELEASE	1748
E.10 CMD_MSG_SRAM_SET_ADDR	1749
E.11 CMD_MSG_SRAM_READ_ADDR	1749
E.12 CMD_MSG_SRAM_SET_ADDR_OFFSET	1750
E.13 CMD_MSG_SRAM_READ_ADDR_OFFSET	1751
E.14 CMD_MSG_SRAM_WRITE	1751
E.15 CMD_MSG_SRAM_WRITE_INC	1752
E.16 CMD_MSG_SRAM_WRITE_IMM	1753
E.17 CMD_MSG_SRAM_READ	1753
E.18 CMD_MSG_SRAM_READ_INC	1754
E.19 CMD_MSG_SRAM_READ_IMM	1755
E.20 CMD_MSG_SET_ECC_CTRL	1755
E.21 CMD_MSG_READ_ECC_CTRL	1757
E.22 CMD_DEBUG_RESET	1757
E.23 CMD_DEBUG_HALT	1758
E.24 CMD_DEBUG_RUN	1759
E.25 CMD_DEBUG_SET_MASK	1759
E.26 CMD_DEBUG_READ_MASK	1760
E.27 CMD_DEBUG_SET_SELECT	1761
E.28 CMD_DEBUG_READ_SELECT	1762
E.29 CMD_DEBUG_READ_EN	1762
E.30 CMD_DEBUG_READ_CMD	1763

E.31	CMD_DEBUG_READ_CORE	1764
E.32	CMD_GFRC_CLEAR	1765
E.33	CMD_GFRC_READ_LO	1765
E.34	CMD_GFRC_READ_HI	1766
E.35	CMD_GFRC_ENABLE	1767
E.36	CMD_GFRC_DISABLE	1767
E.37	CMD_GFRC_READ_DISABLE	1768
E.38	CMD_GFRC_SET_CORE	1768
E.39	CMD_GFRC_READ_CORE	1769
E.40	CMD_GFRC_READ_HALT	1770
E.41	CMD_GFRC_CLK_ENABLE	1771
E.42	CMD_GFRC_CLK_DISABLE	1771
E.43	CMD_GFRC_READ_CLK_STATUS	1772
E.44	CMD_PDM_SET_PM	1773
E.45	CMD_PDM_READ_PSTATUS	1774
E.46	CMD_GFRC_CLEAR	1775
E.47	CMD_GFRC_READ_LO	1776
E.48	CMD_GFRC_READ_HI	1776
E.49	CMD_GFRC_ENABLE	1777
E.50	CMD_GFRC_DISABLE	1777
E.51	CMD_GFRC_READ_DISABLE	1778
E.52	CMD_PMU_SET_PUCNT	1778
E.53	CMD_PMU_READ_PUCNT	1779
E.54	CMD_PMU_SET_RSTCNT	1779
E.55	CMD_PMU_READ_RSTCNT	1780
E.56	CMD_PMU_SET_PDCNT	1781
E.57	CMD_PMU_READ_PDCNT	1781
E.58	CMD_PMU_SET_PUCNT	1782
E.59	CMD_PMU_READ_PUCNT	1782
E.60	Examples: ARConnect ICI	1783
E.60.1	Scenario 1	1783
E.60.2	Scenario2	1784
E.61	Examples: ARConnect Semaphore Unit	1786
E.61.1	Scenario 1	1786
E.61.2	Scenario 2	1786
E.62	Examples: ICM	1787
E.62.1	Scenario 1	1787
E.63	Examples: ICD	1788
E.63.1	Conditionally Halt Cores Scenario	1788
E.63.2	Selectively Halt Cores Scenario	1788
E.63.3	Selectively Run Cores Scenario	1788
E.63.4	Selectively Reset Cores Scenario	1789
E.64	Examples: GFRC	1790

List of Figures

Figure 4-1 Address Space Model	115
Figure 4-2 64-bit Element As a 64-bit Operand	116
Figure 4-3 32-bit Elements in a 64-bit Operand	116
Figure 4-4 16-bit Elements in a 64-bit Operand	116
Figure 4-5 Register Pair Holding Operands in Little-endian Mode	116
Figure 4-6 Register Pair Holding Operands in Big-endian Mode	117
Figure 4-7 32-Bit Elements in a 32-bit Operand	118
Figure 4-8 16-Bit Elements in a 32-bit Operand	118
Figure 4-9 Expansion of a u6 Literal to a 32-Bit Word (w-shimm)	118
Figure 4-10 Expansion of an s12 Literal to a 32-Bit Word (w-shimm)	118
Figure 4-11 Expansion of a u6 Literal to Half-Word (16-bit, hw-shimm)	118
Figure 4-12 Expansion of an s12 Literal to Half-Word (16-bit, hw-shimm)	118
Figure 4-13 A Half-Word (16-bit) Short Limm Value Replicated to form a 64-Bit Operand	118
Figure 4-14 A word (32-bit) Short-limm or Limm value Replicated to form a 64-Bit Operand	119
Figure 4-15 Two 16-Bit Vectors with Expanded u6 Operands	119
Figure 4-16 Two 16-Bit Vectors with Expanded s12 Operands	119
Figure 4-17 Four 8-Bit Vectors with Expanded u6 Operands	119
Figure 4-18 Four 16-Bit Vectors with Expanded s12 Operands	119
Figure 4-19 Four 8-Bit Vectors with s12 Operands	119
Figure 4-20 Expansion of a w6 in STD	119
Figure 4-21 Expansion of a limm in STD	119
Figure 4-22 Single-Precision Floating-Point Data	120
Figure 4-23 Double-Precision Floating-Point Data	120

Figure 4-24 Double Precision Floating-Point Data	121
Figure 4-25 64-bit Register Data in Byte-Wide Memory, Little-Endian	123
Figure 4-26 64-bit Register Data in Byte-Wide Memory, Big-Endian	123
Figure 4-27 Register containing 32-bit Data	124
Figure 4-28 32-bit Register Data in Byte-Wide Memory, Little-Endian	124
Figure 4-29 32-bit Register Data in Byte-Wide Memory, Big-Endian	124
Figure 4-30 Register Containing 16-Bit Data.	125
Figure 4-31 16-bit Register Data in Byte-Wide Memory, Little-Endian	125
Figure 4-32 16-bit Register Data in Byte-Wide Memory, Big-Endian	125
Figure 4-33 Register Containing 8-Bit Data.	125
Figure 4-34 8-bit Register Data in Byte-Wide Memory	125
Figure 4-35 Register Containing 1-Bit Data.	126
Figure 4-36 Half-word Numbering of a 32-Bit Instruction or Long-immediate Data Item	126
Figure 4-37 Little-endian Offset of each Byte in a 32-Bit Instruction or Immediate	126
Figure 4-38 Big-endian Offset of each Byte in a 32-Bit Instruction or Immediate	126
Figure 4-39 Little-endian Memory Offset of each Byte in a 16-Bit Instruction	127
Figure 4-40 Big-endian Memory Offset of each Byte in a 16-bit Instruction	127
Figure 5-1 Core Register Map Summary	131
Figure 5-2 PCL Register	139
Figure 5-3 ACCH for Double-Precision FPU operations	140
Figure 5-4 ACCL for Double-Precision FPU operations	140
Figure 5-5 ACCL for Single-Precision FPU Operations	140
Figure 5-6 LP_START Register (PC_SIZE == 32)	146
Figure 5-7 LP_START Register (PC_SIZE < 32)	146
Figure 5-8 LP_END Register	147
Figure 5-9 LP_END Register (PC_SIZE < 32)	147
Figure 5-10 IDENTITY Register	148
Figure 5-11 PC Register Addressing Full 32-bit Address Space	150
Figure 5-12 PC Register Addressing a Reduced Address Space (PC_SIZE == 32)	150

Figure 5-13 SEC_STAT	152
Figure 5-14 STATUS32 Register	155
Figure 5-15 STATUS32_P0 Register	163
Figure 5-16 AUX_USER_SP Register	164
Figure 5-17 AUX_IRQ_CTRL Register	165
Figure 5-18 INT_VECTOR_BASE Register (PC_SIZE == 32)	167
Figure 5-19 INT_VECTOR_BASE Register (PC_SIZE < 32)	167
Figure 5-20 INT_VECTOR_BASE_S Register (PC_SIZE == 32)	168
Figure 5-21 INT_VECTOR_BASE_S Register (PC_SIZE < 32)	168
Figure 5-22 AUX_KERNEL_SP Register	169
Figure 5-23 AUX_SEC_U_SP Register	170
Figure 5-24 AUX_SEC_K_SP Register	171
Figure 5-25 AUX_NSEC_SP Register	172
Figure 5-26 AUX_IRQ_ACT Register	173
Figure 5-27 IRQ_PRIORITY_PENDING Register	174
Figure 5-28 AUX_IRQ_HINT Register	175
Figure 5-29 IRQ_PRIORITY Register	176
Figure 5-30 NSC_TABLE_TOP Register when PC_SIZE == 32	178
Figure 5-31 NSC_TABLE_BASE Register when PC_SIZE == 32	179
Figure 5-32 JLI_BASE Register when PC_SIZE == 32	180
Figure 5-33 JLI_BASE Register when PC_SIZE < 32	180
Figure 5-34 LDI_BASE Register when ADDR_SIZE == 32	181
Figure 5-35 LDI_BASE Register when ADDR_SIZE < 32	181
Figure 5-36 EI_BASE Register when PC_SIZE == 32	182
Figure 5-37 EI_BASE Register when PC_SIZE < 32	182
Figure 5-38 ERET Register when PC_SIZE == 32	183
Figure 5-39 ERET Register when PC_SIZE < 32	183
Figure 5-40 ERBTA Register when PC_SIZE == 32	184
Figure 5-41 ERBTA Register when PC_SIZE < 32	184

Figure 5-42 ERSTATUS Register	185
Figure 5-43 ECR Register	186
Figure 5-44 EFA Register	188
Figure 5-45 ERSEC_STAT	189
Figure 5-46 AUX_SEC_EXCEPT	191
Figure 5-47 ICAUSE Register	193
Figure 5-48 IRQ_SELECT Register	194
Figure 5-49 IRQ_ENABLE Register	195
Figure 5-50 IRQ_TRIGGER Register	196
Figure 5-51 IRQ_STATUS Register	197
Figure 5-52 XPU Register	198
Figure 5-53 BTA Register (PC_SIZE == 32)	200
Figure 5-54 BTA Register (PC_SIZE < 32)	200
Figure 5-55 IRQ_PULSE_CANCEL Register	202
Figure 5-56 IRQ_PENDING Register	203
Figure 5-57 User Extension Flags Register, XFLAGS	204
Figure 5-58 BCR_VER Register	206
Figure 5-59 BTA_LINK_BUILD Register	207
Figure 5-60 VECBASE_AC_BUILD Register	208
Figure 5-61 SEC_VECBASE_BUILD Register	209
Figure 5-62 RF_BUILD Register	210
Figure 5-63 MULTIPLY_BUILD Register	212
Figure 5-64 SWAP_BUILD Register	213
Figure 5-65 NORM_BUILD Register	214
Figure 5-66 MINMAX_BUILD Register	215
Figure 5-67 BARREL_BUILD Register	216
Figure 5-68 ISA_CONFIG Build Register	217
Figure 5-69 ARCH_CONFIG Register	220
Figure 5-70 IRQ_BUILD Register	221

Figure 6-1 Fast Interrupt Entry	241
Figure 6-2 Fast Interrupt Exit	241
Figure 6-3 Determine Active Interrupts	261
Figure 6-4 Exception Exit in Normal Mode	293
Figure 6-5 Exception-Exit in SecureShield 2x2 Mode	294
Figure 7-1 Long Immediate source data value	308
Figure 8-1 Branch Conditionally Format	360
Figure 8-2 Branch Unconditional Far Format	360
Figure 8-3 Branch on Compare Register-register Format	361
Figure 8-4 Branch on Compare Register-Immediate Format	362
Figure 8-5 Branch and Link Conditionally Format	363
Figure 8-6 Branch and Link Unconditional Far Format	364
Figure 8-7 Load Register with Offset Format	365
Figure 8-8 Store Register with Offset Format	366
Figure 8-9 Store Immediate with Offset Format	366
Figure 8-10 General Operations Register-Register Format	368
Figure 8-11 General Operations Register with Unsigned 16-bit Immediate Format	369
Figure 8-12 General Operations Register with Signed 12-bit Immediate Format	369
Figure 8-13 General Operations Conditional Register Format	369
Figure 8-14 General Operations Conditional Register with Unsigned 6-bit Immediate	370
Figure 8-15 Load Register-Register Format	373
Figure 8-16 Extension ALU Operation, register-register	380
Figure 8-17 Extension ALU Operation, register with unsigned 6-bit immediate	380
Figure 8-18 Extension ALU Operation, register with signed 12-bit immediate	380
Figure 8-19 Extension ALU Operation, conditional register	380
Figure 8-20 Extension ALU Operation, cc register with unsigned 6-bit immediate	381
Figure 9-1 Compact Move Format	385
Figure 9-2 Compact Load Format	385
Figure 9-3 Compact Load and Subtract Format	386

Figure 9-4 Compact Add Format	386
Figure 9-5 Compact Load and Store Format	387
Figure 9-6 Compact Load Indexed Format	387
Figure 9-7 Indexed Jump and Execute Format	388
Figure 9-8 Load and Add Register-Register Format	389
Figure 9-9 Add/Sub/Shift Register-Immediate Format	390
Figure 9-10 Dual Register with a 3-bit b Register Format	391
Figure 9-11 Dual Register with a 3-bit Biased Signed Integer Format	391
Figure 9-12 Long Immediate Operand Format	392
Figure 9-13 Compact General Operations Register-Register Format	393
Figure 9-14 Compact Single Operand, Jumps, Special Formats	394
Figure 9-15 Compact Zero Operand Instructions	395
Figure 9-16 Compact Load/Store with Offset Format	397
Figure 9-17 Compact Shift/Sub Bit Immediate Format	398
Figure 9-18 Compact Load/Add GP-Relative Format	400
Figure 9-19 Compact Load PCL Relative Format	401
Figure 9-20 Compact Move Immediate Format	402
Figure 9-21 Compact ADD/CMP Immediate Format	403
Figure 9-22 Compact Branch on Compare Register with Zero Format	404
Figure 9-23 Compact Branch Conditionally Format	405
Figure 9-24 Branch Conditionally with cc Field Format	405
Figure 9-25 Compact Branch and Link Unconditionally Format	407
Figure 10-1 Loop Detection and Update Mechanism	626
Figure 10-2 SLEEP Operand	735
Figure 11-1 Example Host Memory Maps, Contiguous Host Memory	779
Figure 11-2 Example Host Memory Maps, Host Memory and Host I/O	779
Figure 13-1 SecureShield Overview	787
Figure 14-1 SEC_BUILD Register	794
Figure 15-1 MPU _INDEX for 16 Regions	805

Figure 15-2 MPU_INDEX for 2 Regions	805
Figure 15-3 MPU_RSTART	807
Figure 15-4 MPU_REND	808
Figure 15-5 MPU_RPER	809
Figure 15-6 MPU_PROBE	811
Figure 16-1 AUX_KEY_DATA Register	817
Figure 16-2 AUX_KEY_UNLOCK Register	818
Figure 17-1 ERP_CTRL Register	820
Figure 17-2 RFERP_STATUS_0 Register	824
Figure 17-3 RFERP_STATUS_1 Register	826
Figure 17-4 ESYN Register	827
Figure 17-5 ERP_BUILD Register	828
Figure 18-1 Example 1: MPU_RDB0 Register	836
Figure 18-2 Example 1: MPU_RDP0 Register	836
Figure 18-3 Example 1: MPU_RDB1 Register	836
Figure 18-4 Example 1: MPU_RDP1 Register	836
Figure 18-5 Example 1: MPU_RDB2 Register	836
Figure 18-6 Example 1: MPU_RDP2 Register	836
Figure 18-7 Overlapping Regions Memory Map	837
Figure 18-8 Example 2: MPU_RDB0 Register	837
Figure 18-9 Example 2: MPU_RDP0 Register	837
Figure 18-10 Example 2: MPU_RDB7 Register	837
Figure 18-11 Example 2: MPU_RDP7 Register	837
Figure 18-12 Example 1: MPU_RDB0 Register	838
Figure 18-13 Example 1: MPU_RDP0 Register	838
Figure 18-14 Example 1: MPU_RDB1 Register	838
Figure 18-15 Example 1: MPU_RDP1 Register	838
Figure 18-16 Example 1: MPU_RDB2 Register	838
Figure 18-17 Example 1: MPU_RDP2 Register	838

Figure 18-18 Overlapping Regions Memory Map	839
Figure 18-19 Example 2: MPU_RDB0 Register	839
Figure 18-20 Example 2: MPU_RDP0 Register	839
Figure 18-21 Example 2: MPU_RDB7 Register	839
Figure 18-22 Example 2: MPU_RDP7 Register	839
Figure 18-23 MPU_EN Register	845
Figure 18-24 MPU_ECR Register	848
Figure 18-25 MPU Region Descriptor Base Registers	850
Figure 18-26 MPU Region Descriptor Permissions Registers	851
Figure 18-27 MPU_BUILD Register	854
Figure 19-1 Stack regions	861
Figure 19-2 Stack Checking Scenarios	862
Figure 19-3 USTACK_TOP when ADDR_SIZE == 32	864
Figure 19-4 USTACK_TOP when ADDR_SIZE < 32	864
Figure 19-5 USTACK_BASE when ADDR_SIZE == 32	865
Figure 19-6 USTACK_BASE when ADDR_SIZE < 32	865
Figure 19-7 KSTACK_TOP when ADDR_SIZE == 32	866
Figure 19-8 KSTACK_TOP, when ADDR_SIZE < 32	866
Figure 19-9 KSTACK_BASE when ADDR_SIZE == 32	867
Figure 19-10 KSTACK_BASE when ADDR_SIZE < 32	867
Figure 19-11 S_USTACK_TOP when ADDR_SIZE == 32	868
Figure 19-12 S_USTACK_TOP when ADDR_SIZE < 32	868
Figure 19-13 S_KSTACK_TOP when ADDR_SIZE == 32	868
Figure 19-14 S_KSTACK_TOP, when ADDR_SIZE < 32	868
Figure 19-15 S_USTACK_BASE when ADDR_SIZE == 32	869
Figure 19-16 S_USTACK_BASE when ADDR_SIZE < 32	869
Figure 19-17 S_KSTACK_BASE when ADDR_SIZE == 32	869
Figure 19-18 S_KSTACK_BASE when ADDR_SIZE < 32	869
Figure 19-19 STACK_REGION_BUILD Register	870

Figure 19-20 CPROT_BUILD Register	874
Figure 20-1 AUX_ICCM, base address for ICCM (ADDR_SIZE == 32)	876
Figure 20-2 AUX_ICCM, base address for ICCM (ADDR_SIZE == 16)	876
Figure 20-3 AUX_ICCM, base address for ICCM (ADDR_SIZE == m)	876
Figure 20-4 CCM_PART_SIZE Register	877
Figure 20-5 ICCM_BUILD Register	878
Figure 21-1 IC_IVIC Register	882
Figure 21-2 IFQUEUE_BUILD Register	883
Figure 22-1 IC_IVIC Register	887
Figure 22-2 IC_CTRL Register	888
Figure 22-3 IC_LIL Register	890
Figure 22-4 IC_IVIR Register	892
Figure 22-5 IC_ENDR Register	893
Figure 22-6 IC_IVIL Register	894
Figure 22-7 IC_RAM_ADDR Register for Direct-Cache-Access (AT==0)	896
Figure 22-8 IC_TAG Register for Reads	900
Figure 22-9 IC_TAG Register for Writes	900
Figure 22-10 IC_XTAG Register	904
Figure 22-11 IC_DATA Register	905
Figure 22-12 IC_PTAG Register	908
Figure 22-13 I_CACHE_BUILD Register	909
Figure 23-1 AUX_DCCM, base address for DCCM (ADDR_SIZE == 32)	912
Figure 23-2 AUX_DCCM, base address for DCCM (ADDR_SIZE == 16)	912
Figure 23-3 AUX_DCCM, base address for DCCM (ADDR_SIZE == m)	912
Figure 23-4 DCCM_BUILD Register	913
Figure 24-1 DC_IVDC Register	917
Figure 24-2 DC_CTRL Register	919
Figure 24-3 DC_LDL Register	921
Figure 24-4 DC_IVDL Register	924

Figure 24-5 DC_FLSH Register	926
Figure 24-6 DC_FLDL Register	928
Figure 24-7 DC_STARTR Register	930
Figure 24-8 DC_ENDR Register	931
Figure 24-9 DC_RAM_ADDR for Cache Controlled RAM Access (AT==1)	932
Figure 24-10 DC_RAM_ADDR Register for Direct-Cache-Access (AT==0)	932
Figure 24-11 DC_TAG for Reads	934
Figure 24-12 DC_TAG for Writes	934
Figure 24-13 DC_XTAG Register	936
Figure 24-14 DC_DATA	937
Figure 24-15 AUX_CACHE_LIMIT, start of non-cached regions (ADDR_SIZE == 32)	938
Figure 24-16 AUX_CACHE_LIMIT, (ADDR_SIZE == 16)	938
Figure 24-17 AUX_CACHE_LIMIT, start of non-cached regions (ADDR_SIZE == m)	938
Figure 24-18 D_CACHE_BUILD	940
Figure 25-1 JLAT with JOB_BASE	944
Figure 25-2 JOB_BASE1 Register	945
Figure 25-3 JOB_BASE2 Register	946
Figure 25-4 TASK_SWITCH_BUILD Register	947
Figure 26-1 OLM_VA_START Register	950
Figure 26-2 OLM_VA_END Register	951
Figure 26-3 Page Descriptor Format for 4KB Page Size	952
Figure 26-4 Page Descriptor Format for 8KB Page Size	952
Figure 26-5 Page Descriptor Format for 16KB Page Size	952
Figure 26-6 TLB Page Descriptor Register for 4KB Page Size	954
Figure 26-7 TLB Page Descriptor Register for 8KB Page Size	954
Figure 26-8 TLB Page Descriptor Register for 16KB Page Size	954
Figure 26-9 TLBIndex Register	957
Figure 26-10 TLBCommand Register	959
Figure 26-11 TLBEnable Register	966

Figure 26-12 SCRATCH_DATA0 Register	967
Figure 26-13 OLM_BUILD Register	968
Figure 27-1 NV_ICCM_CTRL Register	972
Figure 27-2 NV_ICCM_ERASE Register	974
Figure 27-3 NV_ICCM_PROG Register	975
Figure 27-4 NV_ICCM_DATA Register	976
Figure 28-1 KEY_STORE_CTRL Register	979
Figure 28-2 KEY_STORE_PWD Register	981
Figure 28-3 KEY_STORE_ADDR Register	982
Figure 28-4 KEY_STORE_DATA Register	983
Figure 28-5 KEY_STORE_SID Register	984
Figure 28-6 KEY_STORE_BUILD Register	985
Figure 29-1 CAL_STORE_CTRL Register	988
Figure 29-2 CAL_STORE_ADDR Register	990
Figure 29-3 CAL_STORE_DATA Register	991
Figure 29-4 CAL_STORE_BUILD Register	992
Figure 30-1 Interrupt Generated after Timer Reaches Limit Value	994
Figure 30-2 Timer 0 Count Value Register	996
Figure 30-3 Timer 0 Control Register	997
Figure 30-4 Timer 0 Limit Value Register	999
Figure 30-5 Timer 1 Count Value Register	1000
Figure 30-6 Timer 0 Control Register	1001
Figure 30-7 Timer 1 Limit Value Register	1002
Figure 30-8 Secure Timer 0 Count Value Register	1003
Figure 30-9 Secure Timer 0 Control Register	1004
Figure 30-10 Secure Timer 0 Limit Value Register	1005
Figure 30-11 Secure Timer 1 Count Value Register	1006
Figure 30-12 Secure Timer 1 Control Register	1007
Figure 30-13 Secure Timer 1 Limit Value Register	1008

Figure 30-14 AUX_RTC_CTRL Register	1009
Figure 30-15 State Machine of A1 and A0 bits in RTC Control Register	1010
Figure 30-16 RTC Count Low Register	1011
Figure 30-17 AUX_RTC_HIGH Register	1012
Figure 30-18 TIMER_BUILD Register	1013
Figure 31-1 EWDT_PASSWD Register	1017
Figure 31-2 EWDT_ESR Register	1018
Figure 31-3 EWDT_INDEX Register	1019
Figure 31-4 EWDT_CTRLx Register	1020
Figure 31-5 EWDT_PERIODx Register	1022
Figure 31-6 EWDT_COUNTx Register	1023
Figure 31-7 EWDT_BUILD Register	1024
Figure 31-8 Timer Configured with No Window, Countdown Timer Mode	1025
Figure 32-1 DMP_PER_AUX, start of peripheral region.	1028
Figure 32-2 Peripheral Regions Based on Base and Limit Values	1029
Figure 32-3 DMP_PP_BUILD Register	1030
Figure 33-1 DEBUG Register	1032
Figure 33-2 DEBUGI Register	1036
Figure 33-3 ACG_CTRL Register	1038
Figure 34-1 Actionpoint Build Register	1044
Figure 34-2 Actionpoint 0 Match Value Register	1045
Figure 34-3 Actionpoint 0 Match Mask Register	1046
Figure 34-4 Actionpoint 0 Control Register	1047
Figure 34-5 Actionpoint 1 Match Value Register	1052
Figure 34-6 Actionpoint 1 Match Mask Register	1053
Figure 34-7 Actionpoint 1 Control Register	1054
Figure 34-8 Actionpoint 2 Match Value Register	1055
Figure 34-9 Actionpoint 2 Match Mask Register	1056
Figure 34-10 Actionpoint 3 Match Value Register	1058

Figure 34-11 Actionpoint 3 Match Mask Register	1059
Figure 34-12 Actionpoint 3 Control Register	1060
Figure 34-13 Actionpoint 4 Match Value Register	1061
Figure 34-14 Actionpoint 4 Match Mask Register	1062
Figure 34-15 Actionpoint 4 Control Register	1063
Figure 34-16 Actionpoint 5 Match Value Register	1064
Figure 34-17 Actionpoint 5 Match Mask Register	1065
Figure 34-18 Actionpoint 5 Control Register	1066
Figure 34-19 Actionpoint 6 Match Value Register	1067
Figure 34-20 Actionpoint 6 Match Mask Register	1068
Figure 34-21 Actionpoint 6 Control Register	1069
Figure 34-22 Actionpoint 7 Match Value Register	1070
Figure 34-23 Actionpoint 7 Match Mask Register	1071
Figure 34-24 Actionpoint 7 Control Register	1072
Figure 34-25 Actionpoint Isolation Enable Register	1073
Figure 34-26 AP_WP_PC, Watchpoint Program Counter Register when PC_SIZE = 32.....	1074
Figure 34-27 AP_WP_PC Register when PC_SIZE < 32	1074
Figure 34-28 Smart Operation	1075
Figure 34-29 SMART_BUILD Value.....	1077
Figure 34-30 SMART_CONTROL Register.....	1078
Figure 34-31 SMART_DATA Register.....	1080
Figure 34-32 SRC_ADDR Value	1080
Figure 34-33 DEST_ADDR Value Register	1081
Figure 34-34 FLAGS Value Register.....	1081
Figure 34-35 DB_STATUS Register	1084
Figure 34-36 DB_CMD Register	1085
Figure 34-37 DB_ADDR Register	1086
Figure 34-38 DB_DATA Register	1087
Figure 34-39 DB_RESET Register	1088

Figure 34-40 Debug Interface Timing Diagram	1089
Figure 35-1 CC_INDEX Register.	1094
Figure 35-2 CC_NAME0 Register.	1095
Figure 35-3 CC_NAME1 Register.	1096
Figure 35-4 PCT_COUNTL Register	1097
Figure 35-5 PCT_COUNTH Register	1098
Figure 35-6 PCT_SNAPL Register	1099
Figure 35-7 PCT_SNAPH Register	1100
Figure 35-8 PCT_CONFIG Register	1101
Figure 35-9 PCT_CONTROL Register	1103
Figure 35-10 PCT_INDEX Register	1105
Figure 35-11 PCT_MINMAXL Register	1106
Figure 35-12 PCT_MINMAXH Register	1107
Figure 35-13 PCT_RANGEL Register	1108
Figure 35-14 PCT_RANGEH Register	1109
Figure 35-15 PCT_UFLAGS Register	1110
Figure 35-16 PCT_BUILD Register	1112
Figure 35-17 CC_BUILD Register	1113
Figure 36-1 RTT_ADDRESS Register	1116
Figure 36-2 RTT_DATA Register	1117
Figure 36-3 RTT_COMMAND Register	1118
Figure 36-4 RTT_BUILD Register	1119
Figure 36-5 ARCT_BUILD Register	1122
Figure 36-6 ARCT_PRSEL Register	1124
Figure 36-7 ARCT_TSTAMP_ENABLE Register	1125
Figure 36-8 ARCT_DEBUGGER_MESSAGE_ENABLE Register	1126
Figure 36-9 RTT_PRDCR_BLD_REG Register	1127
Figure 36-10 ARCT_OCM_BASE Register	1129
Figure 36-11 ARCT_OCM_SIZE Register	1130

Figure 36-12 ARCT_OCM_WPTR Register	1131
Figure 36-13 ARCT_TR_STATUS Register	1132
Figure 36-14 ARCT_FLUSH_CTRL Register	1134
Figure 36-15 ARCT_OFFLOAD_CTRL Register	1135
Figure 36-16 PTRN_GEN Register	1136
Figure 36-17 NEXUS_CLK_DIV Register	1137
Figure 36-18 PR_EVTI_REG Register	1138
Figure 36-19 Start Address Register	1142
Figure 36-20 Trace Address Stop Filter Register	1143
Figure 36-21 MEM_WR_FILTER_START Register	1144
Figure 36-22 MEM_WR_FILTER_STOP Register	1145
Figure 36-23 MEM_RD_FILTER_START Register	1146
Figure 36-24 MEM_RD_FILTER_STOP Register	1147
Figure 36-25 Data Filter0 LSW Register	1148
Figure 36-26 Data Filter0 MSW Register	1149
Figure 36-27 Data Filter1 LSW Register	1150
Figure 36-28 Data Filter1 MSW Filter	1151
Figure 36-29 Producer Type Register	1152
Figure 36-30 Filter Source Select Register	1153
Figure 36-31 Trace Filter Control Register	1154
Figure 36-32 Producer Source Enable Register	1156
Figure 36-33 Producer Watchpoint Status Register	1157
Figure 36-34 Producer Watchpoint Enable Register	1159
Figure 36-35 Producer Cross Trigger Enable Register, CTIEN_REG_ADDR	1161
Figure 36-36 Producer ATB SYNC Frame Insertion Register, SYNCRFR_REG_ADDR	1162
Figure 36-37 Producer Trace Attribute Register	1163
Figure 36-38 Producer CoreSight ID Register	1165
Figure 37-1 PDM_PSTAT Register	1168
Figure 37-2 RTT_PDM_PSTAT Register	1170

Figure 37-3 DVFS_PL Register	1171
Figure 37-4 PDM_PMODE Register	1172
Figure 37-5 PDM_DVFS_BUILD Register	1174
Figure 38-1 Core Register Map Summary	1175
Figure 38-2 ACCL for fusedMultiplyAdd and fusedMultiplySubtract Operations	1176
Figure 38-3 FPU_CTRL Register	1178
Figure 38-4 FPU_STATUS Register	1180
Figure 38-5 AUX_DPFP1L, Double-precision Floating Point D1 Register, Lower Half.	1184
Figure 38-6 AUX_DPFP1H, Double-precision Floating Point D1 Register, Higher Half.	1185
Figure 38-7 AUX_DPFP2L, Double-precision Floating Point D2 Register, Lower Half.	1186
Figure 38-8 AUX_DPFP2H, Double-precision Floating Point D2 Register, Higher Half.	1187
Figure 38-9 FPU_BUILD Register	1188
Figure 39-1 AGU Register Data Organization	1193
Figure 39-2 Extended Core Registers when HAS_AGU==true	1195
Figure 39-3 XCCM_BASE Register	1197
Figure 39-4 YCCM_BASE Register	1198
Figure 39-5 XY_BUILD Register	1199
Figure 40-1 AGU_AUX_APx Register	1202
Figure 40-2 AGU_AUX_OSx Register	1207
Figure 40-3 AGU_AUX_MODx Register with Offset Register Increment	1208
Figure 40-4 AGU_AUX_MODx Register with Immediate Offset	1208
Figure 40-5 AGU_AUX_MODx Register with Bit Reverse Increment Using Offset Register.	1208
Figure 40-6 AGU_AUX_MODx Register with Bit Reverse Increment Using Immediate Offset	1208
Figure 40-7 AGU_AUX_MODx Register with Modulo Wrapping Using Offset and Wrap Registers	1209
Figure 40-8 AGU_AUX_MODx Register with Modulo Wrapping Using Immediate Offset and Wrap	1209
Figure 40-9 AGU_AUX_MODx with Modulo Wrapping Using Offset Register and Immediate Wrap	1209
Figure 40-10 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset Register	1209
Figure 40-11 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset Immediate ...	1209
Figure 40-12 AGU_BUILD Register	1214

Figure 41-1 BS_AUX_CTRL Register	1218
Figure 41-2 BS_AUX_ADDR Register	1219
Figure 41-3 BS_AUX_BIT_OS Register	1220
Figure 41-4 BS_AUX_WDATA Register	1221
Figure 41-5 BS_BUILD Register	1222
Figure 43-1 CONNECT_CMD Register	1227
Figure 43-2 CONNECT_WDATA Register	1228
Figure 43-3 CONNECT_READBACK Register	1229
Figure 43-4 CONNECT_SYSTEM_BUILD Register	1230
Figure 43-5 CONNECT_SEMA_BUILD Register	1232
Figure 43-6 CONNECT_MESSAGE_BUILD Register	1233
Figure 43-7 CONNECT_PMU_BUILD Register	1235
Figure 43-8 CONNECT_GFRC_BUILD Register	1236
Figure 43-9 CONNECT_IVC_BUILD Register	1237
Figure 43-10 CONNECT_ICI_BUILD Register	1238
Figure 43-11 CONNECT_ICD_BUILD Register	1239
Figure 43-12 CONNECT_PDM_BUILD Register	1240
Figure 44-1 DMACTRL Register	1247
Figure 44-2 DMACENB Register	1248
Figure 44-3 DMACDSB Register	1249
Figure 44-4 DMACHPRI Register	1250
Figure 44-5 DMACNPRI Register	1251
Figure 44-6 DMACREQ Register	1252
Figure 44-7 DMACSTAT0 Register	1254
Figure 44-8 DMACSTAT1 Register	1255
Figure 44-9 DMACIRQ Register	1257
Figure 44-10 DMACBASE Register for Two DMA Channels	1258
Figure 44-11 DMACBASE Register for Three or Four DMA Channels	1258
Figure 44-12 DMACBASE Register for Five to Eight DMA Channels	1258

Figure 44-13 DMACBASE Register for 16 DMA Channels	1259
Figure 44-14 DMACTRLx Register.....	1261
Figure 44-15 DMASARx Register.....	1267
Figure 44-16 DMADARx Register	1268
Figure 44-17 DMALLPx Register	1270
Figure 44-18 DMACRST Register.....	1273
Figure 44-19 DMAFENB Register.....	1274
Figure 44-20 DMAFDSB Register	1275
Figure 44-21 DMAC_BUILD Register	1276
Figure 46-1 Vector Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==0.....	1287
Figure 46-2 Vector Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==1.....	1287
Figure 46-3 Vector Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==0.....	1287
Figure 46-4 Vector Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==1.....	1287
Figure 46-5 Wide Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==0.....	1287
Figure 46-6 Wide Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==1.....	1288
Figure 46-7 Wide Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==0.....	1288
Figure 46-8 Wide Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==1.....	1288
Figure 47-1 ACC0_LO Register.....	1292
Figure 47-2 ACC0_GLO Register	1293
Figure 47-3 ACC0_HI Register	1294
Figure 47-4 ACC0_GHI Register.....	1295
Figure 47-5 DSP_BFLY0 Register in Little-Endian Mode	1296
Figure 47-6 DSP_BFLY0 Register in Big-Endian Mode.....	1296
Figure 47-7 DSP_FFT_CTRL Register	1297
Figure 47-8 DSP_CTRL Register.....	1298
Figure 47-9 DSP_BUILD Register.....	1300
Figure E-1 CONNECT_READBACK Register Format for CMD_CHECK_CORE_ID Command.....	1741
Figure E-2 CONNECT_READBACK Register Format for CMD_INTRPT_READ_STATUS Command	1744
Figure E-3 CONNECT_READBACK Register Format for CMD_INTRPT_CHECK_SOURCE Command ...	1745

Figure E-4 CONNECT_READBACK Register Format for CMD_SEMA_CLAIM_AND_READ Command . .	1747
Figure E-5 CONNECT_WDATA Register Format for CMD_MSG_SRAM_SET_ADDR Command	1749
Figure E-6 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_ADDR Command . .	1750
Figure E-7 CONNECT_WDATA Register Format for Set-Offset Command	1750
Figure E-8 CONNECT_READBACK Register Format for Read-Offset Command	1751
Figure E-9 CONNECT_WDATA Register Format for CMD_MSG_SRAM_WRITE Command	1752
Figure E-10 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ Command	1754
Figure E-11 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_INC Command . . .	1754
Figure E-12 CONNECT_WDATA Register Format for CMD_MSG_SET_ECC_CTRL Command	1756
Figure E-13 CONNECT_WDATA Register Format for CMD_DEBUG_RESET Command	1757
Figure E-14 CONNECT_READBACK Register Format for CMD_DEBUG_READ_MASK Command	1761
Figure E-15 CONNECT_READBACK Register Format for CMD_DEBUG_READ_SELECT Command	1762
Figure E-16 CONNECT_READBACK Register Format for CMD_DEBUG_READ_EN	1763
Figure E-17 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CMD Command	1764
Figure E-18 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CORE Command	1764
Figure E-19 CONNECT_READBACK Register Format for CMD_GFRC_READ_LO Command	1766
Figure E-20 CONNECT_READBACK Register Format for CMD_GFRC_READ_HI Command	1766
Figure E-21 CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command	1768
Figure E-22 CONNECT_WDATA Register Format for Command CMD_GFRC_SET_CORE	1769
Figure E-23 CONNECT_READBACK Register Format for Command CMD_GFRC_READ_CORE	1770
Figure E-24 CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command	1771
Figure E-25 CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command .	1772
Figure E-26 CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command	1773
Figure E-27 CONNECT_WDAT Register Format for CMD_PDM_SET_PM Command	1773
Figure E-28 CONNECT_READBACK Register Format for CMD_PDM_READ_STATUS Command	1774
Figure E-29 CONNECT_READBACK Register Format for CMD_GFRC_READ_LO Command	1776
Figure E-30 CONNECT_READBACK Register Format for CMD_GFRC_READ_HI Command	1777
Figure E-31 CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command	1778
Figure E-32 CONNECT_WDATA Register Format for CMD_PMU_SET_PUCNT Command	1779

Figure E-33 CONNECT_READBACK Register Format for CMD_PMU_READ_PUCNT Command	1779
Figure E-34 CONNECT_READBACK Register Format for CMD_PMU_SET_RSTCNT Command	1780
Figure E-35 CONNECT_READBACK Register Format for CMD_PMU_SET_PDCNT Command	1781

List of Examples

Example 4-1 Null Instruction Format	128
Example 5-1 Correct LP_COUNT Setup through a Register	138
Example 6-1 Setting Interrupt Priority or Enable (STATUS32.IE) of an Interrupt	237
Example 6-2 PI is the highest priority pending and enabled interrupt, and its priority is W.Regular Interrupt Entry	244
Example 6-3 Regular Interrupt Entry When SecureShield 2+2 Mode is Configured.....	246
Example 6-4 Regular Interrupt Entry.....	251
Example 6-5 Regular Interrupt Exit	254
Example 6-6 Regular Interrupt Exit When SecureShield 2+2 Mode is Configured	256
Example 6-7 Pass Control to an Handler or Perform a Soft Reset.....	265
Example 6-8 Exception Vector Code	265
Example 6-9 Exception in a Delay Slot	295
Example 10-1 To obtain a semaphore using EX.....	545
Example 10-2 To Release Semaphore using ST	546
Example 10-3 Example Loop Code	625
Example 10-4 Use of Zero Overhead Loop.....	628
Example 10-5 Sleep placement in code	738
Example 10-6 Enable Interrupts and Sleep.....	738
Example 10-7 Force a core to enter sleep mode until DMA channel 2 completes its data transfer	773
Example 10-8 WEVT placement in code	774
Example 15-1 MPU Kernel and User Mode Independence - Example.....	798
Example 35-1 Example Operations	1110
Example 40-1 Updating the Address Pointer Value	1205
Example 41-1 BS_AUX_CTRL.BO Bit Behavior	1218

Example 44-1 Linked-list Transfers for Memory-Mapped DMA Channels.	1270
Example E-1 Segment Program of Scenario 1 – Core1	1783
Example E-2 Segment Program of Scenario 1 – Core2	1784
Example E-3 Segment Program of Scenario 2 - Core0, Core1, Core2	1784
Example E-4 Segment Program of Scenario 2 - Core3.....	1785
Example E-5 Segment Program of Scenario 1- Core1	1786
Example E-6 Segment Program of Scenario 2 - Core 2	1786
Example E-7 Segment Program of Scenario 1 - Core1.....	1787
Example E-8 Conditionally Halt Cores Scenario	1788
Example E-9 Selectively Halt Cores Scenario	1788
Example E-10 Selectively Run Cores Scenario	1788
Example E-11 Selectively Reset Cores Scenario	1789
Example E-12 Segment Program of Scenario 1 - Core2.....	1789
Example E-13 Segment Program to Read Value of GFRC	1790
Example E-14 Segment Program of Common Interrupt 1 Generated by Core 0.....	1791

List of Tables

Table 1-1 Auxiliary Register List	79
Table 1-2 Build Configuration Registers	87
Table 2-1 Exception Vectors and Cause Codes.....	91
Table 3-1 ARCV2 Programming Model Configuration Options	105
Table 3-2 ARCV2 Instruction Set Options	108
Table 4-1 Single-Precision Floating-Point Data Field Descriptions.....	120
Table 4-2 Double-Precision Floating-Point Data Field Descriptions.....	121
Table 4-3 Double Precision Floating-Point Data Field Descriptions.....	121
Table 4-4 Instruction Variants.....	127
Table 4-5 Operand Description.....	128
Table 5-1 Core Register Set	132
Table 5-2 16-bit Instruction Register Encoding.....	133
Table 5-3 Current ABI Register Usage.....	134
Table 5-4 Multiple Bank Configuration.....	135
Table 5-5 Baseline Auxiliary Register Set	141
Table 5-6 Auxiliary Register Set	142
Table 5-7 Key to Auxiliary Register Access Permissions for LR and SR Instructions	143
Table 5-8 Zero Overhead Loop Auxiliary Registers (lpc_size > 0)	144
Table 5-9 Indexed Table Auxiliary Register	144
Table 5-10 XPU, User Extension Permission Auxiliary Register	145
Table 5-11 Extension Flags Register, XFLAGS	145
Table 5-12 LP_START Field Description.....	146
Table 5-13 LP_END Field Description.....	147

Table 5-14 IDENTITY Field Descriptions	148
Table 5-15 ARCV2 Addressable Range vs. PC Size	150
Table 5-16 PC Field Description	150
Table 5-17 SEC_STAT Bit-Field Definitions and Read / Write Accessibility	152
Table 5-18 SEC_STAT Bit Debug Access	153
Table 5-19 SEC_STAT Field Description	153
Table 5-20 STATUS32 Bit-Field Definitions, Read / Write Accessibility	156
Table 5-21 STATUS32 Bit-Field Definitions and Read / Write Accessibility when Secure Mode is Configured ..	157
Table 5-22 STATUS32 Bit-Field Definitions and Read / Write Accessibility in Secure Debug Mode	158
Table 5-23 AUX_USER_SP Field Description	164
Table 5-24 AUX_IRQ_CTRL Field Description	165
Table 5-25 INT_VECTOR_BASE Field Description	167
Table 5-26 AUX_IRQ_ACT Field Description	173
Table 5-27 IRQ_PRIORITY_PENDING Field Description	174
Table 5-28 IRQ_PRIORITY Field Description	177
Table 5-29 ECR Field Description	187
Table 5-30 EFA Field Description	188
Table 5-31 ERSEC_STAT Bit-Field Definitions and Read / Write Accessibility	189
Table 5-32 ERSEC_STAT Field Description	190
Table 5-33 AUX_SEC_EXCEPT Field Description	191
Table 5-34 ICAUSE Field Description	193
Table 5-35 IRQ_SELECT Field Description	194
Table 5-36 IRQ_ENABLE Field Description	195
Table 5-37 IRQ_TRIGGER Field Description	196
Table 5-38 IRQ_STATUS Field Description	197
Table 5-39 IRQ_PULSE_CANCEL Field Description	202
Table 5-40 IRQ_PENDING Field Description	203
Table 5-41 Build Configuration Registers	205

Table 5-42 BCR_VER Field Descriptions	206
Table 5-43 BTA_LINK_BUILD Field Descriptions	207
Table 5-44 VECBASE_AC_BUILD Field Descriptions	208
Table 5-45 Value of the CONFIG[1:0] Field based on Configuration Options	208
Table 5-46 SEC_VECBASE_BUILD Field Descriptions	209
Table 5-47 Value of the CONFIG[9:8] Field based on Configuration Options	209
Table 5-48 RF_BUILD Field Descriptions	210
Table 5-49 MULTIPLY_BUILD Field Descriptions	212
Table 5-50 SWAP_BUILD Field Descriptions	213
Table 5-51 NORM_BUILD Field Descriptions	214
Table 5-52 MINMAX_BUILD field descriptions	215
Table 5-53 BARREL_BUILD Field Descriptions	216
Table 5-54 ISA_CONFIG Field Descriptions	217
Table 5-55 ARCH_CONFIG Field Descriptions	220
Table 5-56 IRQ_BUILD Field Descriptions	221
Table 6-1 Overview of Privileges in Kernel and User Modes	224
Table 6-2 Privilege Access of Instructions when the Secure-Mode Option is Configured	226
Table 6-3 Interrupt Configuration Options	231
Table 6-4 Interrupt Allocation	232
Table 6-5 Interrupt/Exception Vectors	264
Table 6-6 Exception Vectors and Cause Codes	268
Table 6-7 EFA and ERET Entries for Exception Types	275
Table 6-8 Exception and Interrupt Exit Modes	292
Table 7-1 Top-level Formats of the ARCV2 Instruction Set	300
Table 7-2 Assignment of top-level instruction formats to major opcodes in each instruction set profile	302
Table 7-3 Summary of 32-bit instruction formats	304
Table 7-4 Summary of 16-bit instruction formats	305
Table 7-5 Key for 32-bit Addressing Modes and Encoding Conventions	306
Table 7-6 Key for 16-bit Addressing Modes and Encoding Conventions	307

Table 7-7 Condition Codes	308
Table 7-8 Delay Slot Modes	309
Table 7-9 Encoding Static Branch Predictions for BRcc, BBIT0, and BBIT1	310
Table 7-10 Address Write-Back Modes	310
Table 7-11 Scaling Shift	311
Table 7-12 Cache Bypass Modes	311
Table 7-13 Load Store Data Sizes	312
Table 7-14 Load Data Extend Mode	312
Table 7-15 Store Data Source Mode	312
Table 7-16 Instruction Syntax Convention	314
Table 7-17 Integer addition, subtraction and comparison operations	316
Table 7-18 Move operations	317
Table 7-19 Bit-wise logical operations	318
Table 7-20 Bit-mask operations	319
Table 7-21 Single Bit Shift and Rotate Operations	319
Table 7-22 Multi-bit Shift Operations	320
Table 7-23 Shift-assist operations	320
Table 7-24 Selection operations	321
Table 7-25 Byte-swapping operations	321
Table 7-26 Bit-scanning operations	321
Table 7-27 Relational Comparison Operations	322
Table 7-28 Integer Multiply, MAC and Divide Operations	322
Table 7-29 Dual and Quad Integer Multiply and MAC Operations	323
Table 7-30 Integer Vector ADD, SUB, MPY operations	323
Table 7-31 Memory Access Instructions	324
Table 7-32 PREFETCH Instruction Encodings	329
Table 7-33 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions	329
Table 7-34 Auxiliary Register Operations	330
Table 7-35 Control Flow Operations	331

Table 7-36 Delay Slot Execution Modes.	332
Table 7-37 Branch on Compare/Test Mnemonics	334
Table 7-38 Branch on Compare Pseudo Mnemonics, Register-Register.	334
Table 7-39 Branch on Compare Pseudo Mnemonics, Register-Immediate	334
Table 7-40 Delay Slot Execution Modes.	335
Table 7-41 Special Instructions	336
Table 7-42 Floating-Point Unit Instructions, 0x06	340
Table 7-43 Instruction Mnemonics	343
Table 7-44 Basic Saturating Arithmetic Operations.	344
Table 7-45 Vector Unpacking Operations	345
Table 7-46 Vector ALU Operations	347
Table 7-47 Accumulator Operations	348
Table 7-48 Vector 16x16 MAC Operations	348
Table 7-49 Dual 16x16 MAC Operations.	350
Table 7-50 32x32 and 16x16 MAC Operations	350
Table 7-51 Dual 16x8 MAC Operations.	352
Table 7-52 32x16 MAC Operations.	352
Table 7-53 Complex Operations	354
Table 7-54 Vector 64-Bit Operations.	356
Table 7-55 Divide and Square-Root Operations.	356
Table 7-56 Address Generation Instructions.	357
Table 8-1 Branch on Compare/Bit Test Register-Register	362
Table 8-2 Hierarchical Sub-formats of the General Operations Formats	367
Table 8-3 Operand Sub-format Indicators.	368
Table 8-4 Opcode assignment for DOP instructions in major op-code 0x04 (F32_GEN4).	371
Table 8-5 Special DOP sub-formats in F32_GEN4	372
Table 8-6 Opcode assignment for SOP instructions in major op-code 0x04 (F32_GEN4)	374
Table 8-7 Opcode assignment for ZOP instructions in major op-code 0x04 (F32_GEN4)	374
Table 8-8 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5)	375

Table 8-9 Opcode assignment for SOP instructions in major op-code 0x05 (F32_EXT5)	376
Table 8-10 Opcode assignment for ZOP instructions in major op-code 0x05 (F32_EXT5).	377
Table 8-11 Opcode assignment for DOP instructions in major op-code 0x06 (F32_EXT6)	378
Table 8-12 Opcode assignment for SOP instructions in major op-code 0x06 (F32_EXT6).	378
Table 8-13 Opcode assignment for ZOP instructions in major op-code 0x06 (F32_EXT6).	379
Table 8-14 List of FastMath extension pack instructions	381
Table 9-1 16-Bit Indexed Jump and Link Operations	388
Table 9-2 16-Bit, LD / ADD Register-Register	389
Table 9-3 16-Bit, ADD/SUB Register-Immediate.	390
Table 9-4 16-Bit MOV/CMP/ADD with High Register	392
Table 9-5 DOP_format 16-Bit General Operations.	393
Table 9-6 16-Bit Single Operand Instructions	395
Table 9-7 16-Bit Zero Operand Instructions	396
Table 9-8 Summary of 16-Bit Load and Store Instructions with Offset	397
Table 9-9 16-Bit Shift/SUB/Bit Immediate	398
Table 9-10 Opcodes for Stack-based Operations	399
Table 9-11 16-Bit GP Relative Instructions	400
Table 9-12 16-Bit ADD/CMP Immediate	403
Table 9-13 16-Bit Branch on Compare	404
Table 9-14 16-Bit Branch, Branch Conditionally.	405
Table 9-15 16-Bit Branch Conditionally	406
Table 10-1 List of Instructions	410
Table 10-2 Delay Slot Modes for the Jump and Link Instructions.	603
Table 10-3 When Reads and Writes Take Effect.	627
Table 10-4 PREFETCH Instruction Encodings	686
Table 10-5 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions.	687
Table 10-6 SLEEP Operand	735
Table 11-1 Host Accesses to the ARCV2-based processor.	780
Table 11-2 Single Step Flags in Debug Register	781

Table 13-1 Core Modes and Resource Permissions	790
Table 13-2 ARCV2 Instruction Set Options	792
Table 14-1 SEC_BUILD Field Description.	794
Table 15-1 ARCV2 Instruction Set Options	798
Table 15-2 MPU Registers	802
Table 15-3 MPU_INDEX Field Description.	805
Table 15-4 MPU_RPER Field Description.	809
Table 15-5 MPU_PROBE Field Description	811
Table 16-1 Secure Debug Registers.	816
Table 17-1 Error Protection Auxiliary Registers.	819
Table 17-2 ERP_CTRL Field Descriptions	820
Table 17-3 ERP_CTRL Fault Bit Descriptions.	822
Table 17-4 RFERP_STATUS_0 Field Description.	824
Table 17-5 RFERP_STATUS_1 Field Description.	826
Table 17-6 ESYN Field Description	827
Table 17-7 ERP_BUILD Register.	828
Table 18-1 Example 1 -- Contiguous Regions Memory Map	835
Table 18-2 Priority Configurations.	840
Table 18-3 Setting ECR Register based on EV_ProtV Exception	841
Table 18-4 ECR parameter values for EV_ProtV sources	842
Table 18-5 MPU Register Set.	843
Table 18-6 MPU Enable Register	845
Table 18-7 MPU Exception Cause Register	849
Table 18-8 MPU Region Descriptor Base Registers Field Description	850
Table 18-9 MPU Region Descriptor Permission Registers Field Description	852
Table 18-10 Build Configuration Registers	853
Table 18-11 MPU_BUILD Field Description.	854
Table 19-1 Stack Checking Specification	862
Table 19-2 Stack Region Checking Auxiliary Registers.	863

Table 19-3 STACK_REGION_BUILD Field Descriptions	870
Table 19-4 CPROT_BUILD Field Descriptions	874
Table 20-1 ICCM Registers	875
Table 20-2 CCM_PART_SIZE Field Descriptions	877
Table 20-3 ICCM_BUILD Field Descriptions	878
Table 21-1 IFQ Auxiliary Registers	881
Table 21-2 IFQUEUE_BUILD Field Descriptions	883
Table 22-1 Basic Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==0)	885
Table 22-2 Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==1)	886
Table 22-3 Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==2)	886
Table 22-4 IC_IVIC Description When SecureShield 2+2 mode is Configured	887
Table 22-5 IC_CTRL Control Flags	888
Table 22-6 IC_LIL Description	891
Table 22-7 IC_IVIL Description	894
Table 22-8 IC_RAM_ADDR for Cache Controlled RAM Access (AT==1)	896
Table 22-9 IC_RAM_ADDR for Aliased Configurations	896
Table 22-10 IC_RAM_ADDRESS Register Behavior	897
Table 22-11 Cache Line Access in the Debug Unlocked in the Normal Mode	899
Table 22-12 IC_TAG Register for Reads in an Aliased Configuration	900
Table 22-13 IC_TAG Register Behavior	901
Table 22-14 IC_DATA Register Behavior	906
Table 22-15 I_CACHE_BUILD Field Descriptions	909
Table 23-1 DCCM Auxiliary Registers	911
Table 23-2 DCCM_BUILD Field Descriptions	913
Table 24-1 Basic Data Cache Auxiliary Registers (DC_FEATURE_LEVEL==0)	915
Table 24-2 Data Cache Lock/Flush Registers (DC_FEATURE_LEVEL > 0)	915
Table 24-3 Data Cache Auxiliary Registers for Direct RAM Access (DC_FEATURE_LEVEL >1)	916
Table 24-4 DC_IVDC Field Descriptions	917
Table 24-5 DC_IVDC Description When SecureShield 2+2 Option is Configured	918

Table 24-6 DC_CTRL Control Flags	919
Table 24-7 DC_LDLD Description When SecureShield 2+2 mode Option is Configured	922
Table 24-8 DC_IVDL Description When SecureShield 2+2 mode Option is Configured	925
Table 24-9 DC_FLSH Control Flags	926
Table 24-10 DC_FLSH Description When the Enhanced Security Package is Configured	927
Table 24-11 DC_FLDL Description When SecureShield 2+2 mode is Configured	929
Table 24-12 Cache Line Access in the Debug Unlocked in the Normal Mode	932
Table 24-13 DC_TAG Control Flags and Update Conditions	934
Table 24-14 Build Configuration Registers	939
Table 24-15 D_CACHE_BUILD Field Descriptions	940
Table 25-1 Task Switch Auxiliary Registers	943
Table 25-2 JOB_BASE1 Field Descriptions	945
Table 25-3 JOB_BASE2 Field Descriptions	946
Table 25-4 TASK_SWITCH_BUILD Field Descriptions	947
Table 26-1 Special Purpose Registers for TLB Control	949
Table 26-2 TLBIndex Addresses	957
Table 26-3 Result Code, Read Only	958
Table 26-4 TLB Command Register Command List	959
Table 26-5 OLM_BUILD Register Fields	968
Table 27-1 NV_ICCM Registers	971
Table 27-2 NV_ICCM_CTRL Field Descriptions	972
Table 27-3 NV_ICCM_ERASE Field Descriptions	974
Table 28-1 Calibration Storage Registers	977
Table 28-2 KEY_STORE_CTRL Field Descriptions	979
Table 28-3 KEY_STORE_PWD Field Descriptions	981
Table 28-4 KEY_STORE_BUILD	985
Table 29-1 Calibration Storage Registers	987
Table 29-2 CAL_STORE_CTRL Field Descriptions	988
Table 29-3 CAL_STORE_BUILD	992

Table 30-1 Auxiliary Registers for Hardware Counter 0 (has_timer_0 == true)	994
Table 30-2 Auxiliary Registers for Hardware Counter 1 (has_timer_1 == true)	995
Table 30-3 Auxiliary Register for Real-time counter	995
Table 30-4 Auxiliary Registers for Hardware Counter 0 (-sec_timer_0==true)	995
Table 30-5 Auxiliary Registers for Hardware Counter 0 (-sec_timer_1==true)	995
Table 30-6 TIMER 0 Field Descriptions	997
Table 30-7 RTC Control Register	1009
Table 30-8 TIMER_BUILD Field Descriptions	1013
Table 31-1 AUX Direct + Index	1016
Table 31-2 EWDT_PASSWORD Field Description	1017
Table 31-3 EWDT_ESR Bit Definition	1018
Table 31-4 EWDT_INDEX Field Definition.	1019
Table 31-5 EWDT_CTRLx Bit Definition.	1020
Table 31-6 EWDT_BCR Bit Definition	1024
Table 32-1 DMP Registers	1027
Table 32-2 DMP_PER_AUX Field Descriptions	1029
Table 32-3 DMP_PP_BUILD Field Descriptions.	1030
Table 33-1 Host Debug Interface Auxiliary Registers	1031
Table 33-2 DEBUG Register Field Description	1032
Table 33-3 Internal Clock During Sleep Mode	1035
Table 33-4 DEBUGI Register Field Description	1036
Table 33-5 ACG_CTRL Field Description	1038
Table 34-1 Auxiliary Registers for Actionpoints	1040
Table 34-2 State of Auxiliary Registers upon Reset	1042
Table 34-3 Actionpoint Build Field Descriptions	1044
Table 34-4 Actionpoint Target Field, AT, Type	1047
Table 34-5 Transaction Type Field, TT, Type	1048
Table 34-6 Mode Field, M, Type	1048
Table 34-7 Pair Field, P, Type	1049

Table 34-8 Actionpoint Action Field, AA Field, Type	1050
Table 34-9 Quad Type	1051
Table 34-10 Actionpoint Isolation Enable Field Descriptions	1073
Table 34-11 SMART_BUILD	1077
Table 34-12 SMART_CONTROL Register	1078
Table 34-13 SMART_DATA Register	1080
Table 34-14 SRC_ADDR Value	1081
Table 34-15 DEST_ADDR Register	1081
Table 34-16 FLAGS Value Register	1081
Table 34-17 Debug Register Addresses	1083
Table 34-18 DB_STATUS Field Description	1084
Table 34-19 DB_CMD Field Description	1085
Table 34-20 DB_ADDR Field Description	1086
Table 34-21 DB_DATA Field Description	1087
Table 34-22 DB_ADDR Field Description	1088
Table 35-1 Performance Auxiliary Registers	1092
Table 35-2 Countable-Conditions Registers	1093
Table 35-3 CC_INDEX Field Description	1094
Table 35-4 CC_NAME0 Field Description	1095
Table 35-5 CC_NAME1 Field Description	1096
Table 35-6 PCT_CONFIG Field Description	1101
Table 35-7 PCT_CONTROL Field Description	1103
Table 35-8 PCT_INDEX Field Description	1105
Table 35-9 PCT_UFLAGS Field Description	1111
Table 35-10 PCT_BUILD Field Description	1112
Table 35-11 CC_BUILD Field Description	1113
Table 36-1 ARC Trace Registers	1115
Table 36-2 RTT_COMMAND Bit Field Description	1118
Table 36-3 RTT_BUILD Field Descriptions	1119

Table 36-4 ARC Trace Address Map	1120
Table 36-5 ARC Trace Control Registers	1121
Table 36-6 ARCT_BUILD Field Descriptions	1122
Table 36-7 ARCT_PRSEL Field Descriptions	1124
Table 36-8 ARCT_TSTAMP_ENABLE Field Descriptions	1125
Table 36-9 ARCT_DEBUGGER_MESSAGE_ENABLE Field Descriptions	1126
Table 36-10 RTT_PRDCR_BLD_REG Description	1127
Table 36-11 ARC Trace Configuration Registers	1127
Table 36-12 ARCT_OCM_BASE Field Descriptions	1129
Table 36-13 ARCT_OCM_SIZE Field Descriptions	1130
Table 36-14 ARCT_OCM_WPTR Field Descriptions	1131
Table 36-15 ARCT_TR_STATUS Field Descriptions	1132
Table 36-16 ARCT_FLUSH_CTRL Field Descriptions	1134
Table 36-17 ARCT_OFFLOAD_CTRL Field Descriptions	1135
Table 36-18 PTRN_GEN Field Descriptions	1136
Table 36-19 NEXUS_CLK_DIV Field Descriptions	1137
Table 36-20 PR_EVTI_REG Field Descriptions	1138
Table 36-21 ARC Core Producer Registers	1138
Table 36-22 Start Address Register and Stop Register Address Offsets	1140
Table 36-23 Start Address Register Field Descriptions	1142
Table 36-24 Stop Address Register Field Descriptions	1143
Table 36-25 MEM_WR_FILTER_START Field Descriptions	1144
Table 36-26 MEM_WR_FILTER_STOP Field Descriptions	1145
Table 36-27 MEM_RD_FILTER_START Field Descriptions	1146
Table 36-28 MEM_RD_FILTER_STOP	1147
Table 36-29 Data Filter0 LSW Description	1148
Table 36-30 Data Filter0 MSW Field Descriptions	1149
Table 36-31 Data Filter1 LSW Field Descriptions	1150
Table 36-32 Data Filter1 MSW Description	1151

Table 36-33 Producer Type Register Field Descriptions	1152
Table 36-34 Filter Source Select Register Description	1153
Table 36-35 Trace Filter Control Register	1154
Table 36-36 Producer Source Enable Description	1156
Table 36-37 Producer Source Watchpoint Status Register Description	1157
Table 36-38 Producer Watchpoint Enable Register Description	1159
Table 36-39 Producer Cross Trigger Enable Register Description.....	1161
Table 36-40 Producer SYNC Frame Insertion Register Description	1162
Table 36-41 Producer Trace Attribute Register Description.....	1163
Table 36-42 Producer CoreSight ID Register Description.....	1165
Table 37-1 Power Domain Management Registers	1167
Table 37-2 PDM_PSTAT Field Description.....	1168
Table 37-3 RTT_PDM_PSTAT Field Description	1170
Table 37-4 DVFS_PL Field Description	1171
Table 37-5 PDM_PMODE Field Description.....	1172
Table 37-6 PDM_DVFS_BUILD Field Description.....	1174
Table 38-1 FPU Registers	1177
Table 38-2 FPU_CTRL Field Description.....	1178
Table 38-3 FPU_STATUS Field Description	1180
Table 38-4 AUX_DPFP1L Field Description.....	1184
Table 38-5 AUX_DPFP1H Field Description	1185
Table 38-6 AUX_DPFP2L Field Description.....	1186
Table 38-7 AUX_DPFP2H Field Description	1187
Table 38-8 FPU_BUILD Field Descriptions.....	1188
Table 39-1 XY Registers	1196
Table 39-2 XY_BUILD Field Descriptions	1199
Table 40-1 AGU Registers	1201
Table 40-2 Number of Address Pointer Registers	1202
Table 40-3 Post-Incrementing Address Pointer Modes.....	1203

Table 40-4 AGU_AUX_APx Field Description	1204
Table 40-5 Number of Offset or Wrap Registers	1207
Table 40-6 Number of Modifier Registers	1208
Table 40-7 Data Transformation-Related Fields in the Modifier Registers	1212
Table 40-8 Data Transformation Modes Based on the FX Field	1212
Table 40-9 AGU_BUILD Field Descriptions	1215
Table 41-1 Bitstream Auxiliary Registers	1217
Table 41-2 BS_AUX_CTRL Field Description	1218
Table 41-3 BS_AUX_ADDR Field Description	1219
Table 41-4 BS_AUX_BIT_OS Field Description	1220
Table 41-5 BS_AUX_WDATA Field Description	1221
Table 41-6 Build Configuration Registers	1221
Table 41-7 BS_BUILD Field Descriptions	1222
Table 43-1 ARConnect Registers	1225
Table 43-2 CONNECT_CMD Register Field Description	1227
Table 43-3 CONNECT_WDATA Register Field Description	1228
Table 43-4 CONNECT_READBACK Register Field Description	1229
Table 43-5 CONNECT_SYSTEM_BUILD Register Field Description	1230
Table 43-6 CONNECT_SEMA_BUILD Register Field Description	1232
Table 43-7 CONNECT_MESSAGE_BUILD Register Field Description	1233
Table 43-8 CONNECT_PMU_BUILD Field Description	1235
Table 43-9 CONNECT_GFRC_BUILD Field Description	1236
Table 43-10 CONNECT_IVC_BUILD Field Description	1237
Table 43-11 CONNECT_ICI_BUILD Field Description	1238
Table 43-12 CONNECT_ICD_BUILD Field Description	1239
Table 43-13 CONNECT_PDM_BUILD Field Description	1240
Table 44-1 DMA Auxiliary Registers	1243
Table 44-2 XY DMA Controller Auxiliary Registers	1245
Table 44-3 DMACTRL Field Description	1247

Table 44-4 DMACENB Field Description	1248
Table 44-5 DMACDSB Field Description	1249
Table 44-6 DMACHPRI Field Description	1250
Table 44-7 DMACNPRI Field Description	1251
Table 44-8 DMACREQ Field Description	1252
Table 44-9 DMACSTAT0 Field Description	1254
Table 44-10 DMACSTAT1 Field Description	1255
Table 44-11 DMACIRQ Field Description	1257
Table 44-12 Memory-Mapped DMA Channel-Specific Registers	1259
Table 44-13 DMAC Channel Auxiliary Registers	1260
Table 44-14 DMAC Channel Registers	1260
Table 44-15 DMACTRLx OP Field Description	1261
Table 44-16 DMACTRLx RT Field Description	1263
Table 44-17 DMACTRLx DTT Field Description	1263
Table 44-18 DMACTRLx DW Field Description	1264
Table 44-19 DMACTRLx BLOCK_SIZE Field Description	1264
Table 44-20 DMACTRLx ARB Field Description	1265
Table 44-21 DMACTRLx I Bit Description	1265
Table 44-22 DMACTRLx AM Field Description	1266
Table 44-23 DMAC Channel Descriptors	1270
Table 44-24 Initial State of DMA Channel Descriptor Registers for Example 44-1	1270
Table 44-25 DMAC Channel Descriptors	1271
Table 44-26 Final State of DMA Channel Descriptor Registers for Example 44-1	1271
Table 44-27 DMAC_BUILD Field Descriptions	1276
Table 45-1 DSP Configuration Options	1282
Table 46-1 8-Bit Signed Integer Vector	1283
Table 46-2 16-Bit Signed Integer	1283
Table 46-3 16-Bit Unsigned Integer	1284
Table 46-4 16-Bit Signed Integer Vector	1284

Table 46-5 16-Bit Unsigned Integer Vector	1284
Table 46-6 16-Bit Signed Fractional Vector	1285
Table 46-7 Complex 16+16 Bit Complex Signed Fractional Data	1285
Table 46-8 32-Bit Signed Integer	1285
Table 46-9 32-Bit Unsigned Integer	1285
Table 46-10 32-Bit Signed Fractional Data	1286
Table 46-11 Accumulator Instructions	1288
Table 47-1 DSP Auxiliary Registers	1291
Table 47-2 ACC0_GLO Field Descriptions	1293
Table 47-3 ACC0_GHI Field Descriptions	1295
Table 47-4 DSP_FFT_CTRL Field Descriptions	1297
Table 47-5 DSP_CTRL Field Descriptions	1298
Table 47-6 DSP_BUILD Field Descriptions	1300
Table 48-1 List of Instructions	1301
Table A-1 Baseline Differences: ARC 600, ARC 700, and ARCV2	1722
Table A-2 ISA Option Differences: ARC 600, ARC 700, and ARCV2	1724
Table A-3 ISA Features: ARC EM and ARC HS	1726
Table B-1 Implementation Behavior on Division Overflow	1730
Table E-1 CONNECT_READBACK Register Field Descriptions for CMD_CHECK_CORE_ID Command ..	1742
Table E-2 CONNECT_READBACK Register Field Descriptions for CMD_INTERRUPT_READ_STATUS Com- mand	1744
Table E-3 CONNECT_READBACK Register Field Descriptions for CMD_INTRPT_CHECK_SOURCE Com- mand	1745
Table E-4 CONNECT_READBACK Register Field Descriptions for CMD_SEMA_CLAIM_AND_READ Com- mand	1747
Table E-5 CONNECT_WDATA Register Field Description for CMD_MSG_SET_ECC_CTRL	1756
Table E-6 CONNECT_WDATA Field Descriptions for CMD_DEBUG_RESET Command	1758
Table E-7 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_EN Command ..	1763
Table E-8 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_CORE Command ... 1764	
Table E-9 CONNECT_WDATA Field Descriptions	1769

Table E-10 CONNECT_WDATA Register Field Descriptions for CMD_PDM_SET_PM Command 1774

Table E-11 CONNECT_READBACK Register Field Descriptions for CMD_PDM_READ_STATUS Command . . 1775

Preface

Web Resources

The following web links are various Synopsys online resources you may find useful:

- DesignWare IP product information: <https://www.synopsys.com/designware-ip.html>
- Your custom DesignWare IP page: <https://www.synopsys.com/dw/mydesignware.php>
- Documentation through SolvNetPlus: <https://www.solvnetplus.com> (Synopsys password required)
- Synopsys Common Licensing (SCL): <https://www.synopsys.com/keys>

Accessing SolvNetPlus

SolvNetPlus includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNetPlus also gives you access to a wide range of Synopsys online services, including downloading software, viewing documentation on the Web, and making a call to the Synopsys Technical Support Center.

To access SolvNetPlus:

1. Go to the SolvNetPlus Web page at <https://solvnetplus.synopsys.com/>.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNetPlus.)

If you need help using SolvNetPlus, click **SolvNetPlus Help** in the **Support Resources** section.

Customer Support

Customer support is available through SolvNetPlus online customer support and through contacting the Synopsys Technical Support Center. For general usage information, see the following article in SolvNetPlus:

<https://solvnetplus.synopsys.com/s/article/SolvNetPlus-Usage-Help-Resources>



Note You must be logged in to SolvNetPlus before clicking this link.

Contacting the Synopsys Technical Support Center

Synopsys provides various methods for contacting the Synopsys Technical Support Center, as follows:

- *For the fastest response, enter a case through SolvNetPlus:*

- a. <https://solvnetplus.synopsys.com>



SolvNetPlus does not support Internet Explorer. Use a supported browser such as Microsoft Edge, Google Chrome, Mozilla Firefox, or Apple Safari.

- b. Click the **Cases** menu and then click **Create a New Case** (below the list of cases).
- c. Complete the mandatory fields marked with an asterisk and click **Save**.

Make sure to include the following:

- **Product L1:** Select one of the following:
 - **DesignWare Cores ARC Tools**
 - **DesignWare Cores ARC Processors**
- **Product L2:** (Select the name that best matches your product)
- d. After creating the case, attach any debug files you created.

SolvNetPlus Online Customer Support

- When you send an e-mail message to support_center@synopsys.com, your email is queued and then, on a first-come, first-served basis, manually routed to the correct support engineer. Kindly note to:
 - Include the full product name and version number in your email so it can be routed correctly.
 - For simulation issues, include the timestamp of any signals or any locations in waveforms that are not understood.
 - Attach any debug files you created.
- Or, telephone your local support center:
 - North America:
Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
 - All other countries:
<https://www.synopsys.com/support/global-support-centers.html>

Accessing ARC Application Notes

SolvNetPlus articles include useful application notes to help you do your job using DesignWare ARC products. To find application notes for your DesignWare ARC product, do the following:

1. Log into SolvNetPlus at <https://solvnetplus.synopsys.com/>.
2. Enter the name of your product in the **Search** field.
3. Click **Search**.

The figure below shows example results.



Note

Application notes are updated frequently, so results can vary.

The screenshot shows the Synopsys SolvNetPlus search interface. The search bar at the top contains 'ARC HS'. Below the search bar, there are navigation tabs: Cases, STARs, Knowledge Base, Legacy Docs Search, and Feedback. The 'Knowledge Base' tab is selected. The search results are displayed under the 'Knowledge' section, showing 5+ results sorted by Relevance. The results list includes:

- Software Bootloading on ARC EM and ARC HS** (000024517) - Last Modified Dec 12, 2019, 7:28 PM. Question: How to initialize ICCM and DCCM from ROM on an ARC EM or ARC HS processor?
- Power Management on ARC HS** (000030883) - Last Modified Dec 25, 2019, 1:54 AM. HS processor family and gives guidance on using them. These features include functional clock.
- ARCV2 and EV Hardware ID Core Versions and Compiler Options** (000025590) - Last Modified Dec 12, 2019, 9:24 PM. 0x43 32 bit 3 stage coao -core3 -av2em ARC HS Note: ARC HS has two versions: ARC HS3x and ARC.
- Useful Signals for Debugging ARC HS3x RTL Simulations** (000025072) - Last Modified Dec 12, 2019, 7:06 PM. HS3x hierarchy for helping debug RTL simulations.
- DesignWare ARC Processor Training** (000015152) - Last Modified Dec 16, 2019, 12:38 AM. pages. Direct links to the forms are listed here: ARC 600 ARC 700 ARC EM ARC HS ARC MQX RTOS.

At the bottom of the search results, there is a link 'Not finding what you are looking for?' and a button 'Create a New Case'.

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IP. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IP implements industry-standard specifications that are currently under review to remove exclusionary language.

Document Revision History

The table below tracks the significant documentation changes that occur from release-to-release and during a release.

Version	Document Date	Description
5755-101	October 2021	■ Added a note to the SMART_CONTROL register. ■ Added the Overlay Management Unit chapter ■ Actionpoint - Added the AP_ISO_EN register that lets you enable or disable actionpoint isolation - Updated the version of the AP_BUILD register to 0x7 ■ SmaRT - Updated the version of the SMART_BUILD register to 0x6 - Added the [NT] to the FLAGS register. This field indicates that the program flow is changed by the next task instruction. ■ MPU: - Updated the version of MPU_BUILD register to 0x8 - Added the ECC field to the MPU_BUILD register - When secure MPU is configured, the kernel mode and user mode are completely independent. That is, user mode permissions do not imply kernel mode permissions when using the secure MPU. See Enabling the Memory-Protection Unit. ■ Interrupts - Added Table 6-4 to describe the interrupt allocation based on the presence of various processor configurations. The change involves in allocation interrupt line 22 to single-bit errors ■ Exceptions; Added the following exceptions: - Memory Read protection violation (0x00060108) - Memory Write Protection Violation (0x00060208) - Memory Read-Modify-Write Protection violation (0x00060308) - Instruction Fetch Protection Violation (0x00060008) - Data TLB Miss (0x00050100, 0x00050200, 0x00050300) - Instruction Fetch TLB Miss (0x00040000) - Fatal TLB Error (0x00030200) - Overlapping TLB Entries (0x00030100) ■ Instruction Cache; added support for region based operations: - Added the IC_IVIR register - Added the IC_ENDR register - Added the IC_PTAG register ■ Data Cache; added support for region-based operations: - Added the DC_STARTR register - Added the DC_ENDR register

Version	Document Date	Description
5755-099	June 2021	■ ICCM: - Updated the version of the ICCM_BUILD register to 0x7 - Added the CCM_PART_SIZE register to support for ICCM0 partitioning ■ Updated the version of the DCCM_BUILD register to 0x7 ■ Updated the version of the BCR_VER register to 0x3 ■ Updated the version of the I_CACHE_BUILD register to 0x7 ■ Updated the version of the D_CACHE_BUILD register to 0x6 ■ ARC Trace - Added register: Trace Producer Type Register, ARCT_PRTYPE register - Updated the version of RTT_BUILD to 0x6 - Removed the freeze feature ■ Error Protection: - Updated the ERP_BUILD version to 0x8 - Updated the description of the ERP_BUILD[IP] field to specify the control for ICCM0 partition - Added field ERP_CTRL[9:8] to support error protection on Overlay Manager JTLB memory ■ PDM: - Updated the version of the PDM_DVFS_BUILD to 0x5 - Added fields: PDM_PMODE[7] and PDM_PSTAT[15] to support ICCM0 partition ■ Task Switching: - Added the chapter: Task Switching Component - Added the NEXTTASK instruction
5755-098	November 2020	■ Added the following registers to support the DesignWare Peripheral flow controller feature: “DMAFENB” and “DMAFDSB”. ■ Removed the safety features.
5755-096	December 2019	■ Updated current version number to 0x6 for “ERP_BUILD” register
5755-091	March 2019	■ Minor grammatical and formatting fixes. ■ Added the AGU_BUILD and DSP_BUILD register entries to Table 1-2 on page 87. ■ Added the EFA_EXT and DVFS_PL register entries to Table 1-1 on page 79. ■ Updated the register layout to match the description of PUC field in “Core Power Status Register, PDM_PSTAT” on page 1168. ■ Added the missing IDU field to the register layout in “CONNECT_SYSTEM_BUILD” on page 1230. ■ Corrected the heading-level of the ARConnect registers.
5755-090	October 2018	■ Updated the cache line sizes in the D_CACHE_BUILD and I_CACHE_BUILD registers.

Version	Document Date	Description
5755-089	September 2018	- Fixed the description of the ERP_CTRL register to match the RTL. - Updated the version number of the SEC_BUILD register to indicate support for the non-maskable imprecise exception. - Updated the EFA register behavior. Unlike previous versions (ARCVVER < 0x44), the EFA register now always exists in the processor.
5755-087	January 2018	Fixed a formatting issue in the <i>Machine Check, Non-Maskable Imprecise Exception, Double fault</i> section.
5755-086	December 2017	- Added support for an external non-maskable imprecise exception. - Updated the EFA register behavior. Unlike previous versions (ARCVVER < 0x44), the EFA register now always exists in the processor.
5755-082	September 2017	- Updated the peripheral bus region description. - Bug Fixes - Clarified the differences between normal and secure interrupt entry and exit. - Explained functions used in pseudo-code. - Removed obsolete ECR values from Table 6-6: Exception Vectors and Cause Codes. - Added Table 6-7: EFA and ERET Entries for Exception Types. - Revised exception entry and exit pseudo-code. - Clarified setting of Z flag in pseudo-code to indicate whether processor was in user or kernel mode when an exception occurred. - Revised list of Exception Types and Priorities. - Documented reset value of MPU Region Permission Register, MPU_RPER. - Clarified MPU_EN reset value and user access permissions. - Clarified behavior of the MPU_INDEX register in combination with MPU_PROBE. - Added more information on switching between operating modes with a Secure MPU.
5755-079	December 2016	ARCV2 ISA updates for ARC EM 4.0 family of processors. Added support for the following secure features: - SecureShield 2+2 modes - Secure MPU - Side-channel protection - ARCV2 ISA alignment - Updates to the XY AGU modifiers - An HRESP error on a peripheral (AHB) bus in ARC EM v4.00a and later results in a Memory Error exception (Vector Number 0x1). Previous versions of ARC EM took an EV_MachineCheck exception (Vector Number 0x3) when a HRESP error occurred. Changes were made to the exception registers to support this behavior. NOTE: When migrating code from previous ARC EM designs (v3.x and earlier) with external AHB bus interfaces to ARC EM v4.0 and later, exception handlers must be modified accordingly.
5755-073	February 2016	ARCV2 ISA updates for ARC EM 3.1 family of processors

Version	Document Date	Description
5755-072	September 2015	ARCV2 ISA updates for ARC EM 3.0 family of processors
5755-046	November 2014	■ Added the DEBUG.EH bit
5755-035	August 2014	ARCV2 ISA for ARC EM 2.0 family of processors: ■ ARCV2DSP (ARCV2ISA extension) with more than 100 DSP instructions ■ Added DSP support ■ Added FPU support ■ Added performance monitor support ■ Added ARC RTT (ARC Real-time Trace) support ■ Added ARConnect (multi-core IP) support
5755-011	October 2012	ARCV2 ISA release for ARC EM family of processors
5755-002	December 2013	Added descriptions for the following registers: ■ Watchdog Timer registers □ WDT_CTRL □ WDT_PASSWD □ WDT_COUNT □ WDT_PERIOD ■ SEP_CTRL and SEP_BUILD registers ■ Updated first technical review comments from R&D
5755-001	October 2013	Initial version

Quick Reference

In this part:

- [Build and Auxiliary Register List](#)
- [Exceptions List](#)

1

Build and Auxiliary Register List

This chapter lists all the auxiliary registers and build configuration registers available in the ARCV2 ISA-based processor. The [Core Architectural State Registers](#) chapter discusses the baseline auxiliary and build configuration registers. The auxiliary and build configuration registers for each system or memory component are discussed in that respective component chapter.

Table 1-1 Auxiliary Register List

Register Address	Register Name
0x02	Loop Start Register, LP_START
0x03	Loop End Register, LP_END
0x04	Core Identity Register, IDENTITY
0x05	Debug Register, DEBUG
0x06	Program Counter, PC
0x09	Secure Status Register, SEC_STAT
0x0A	Status Register, STATUS32
0x0B	Status Register Priority 0, STATUS32_P0
0x0D	Saved User Stack Pointer, AUX_USER_SP
0x0E	Interrupt Context Saving Control Register, AUX_IRQ_CTRL
0x0F	Interrupt and Register Bank Debug Register, DEBUGI
0x10	Invalidate Instruction Cache, IC_IVIC
0x11	Instruction Cache Control Register, IC_CTRL
0x13	Lock Instruction Cache Line, IC_LIL
0x16	Invalidate Instruction-Cache Start Region, IC_IVIR
0x17	Invalidate Instruction-Cache End Region, IC_ENDR
0x18	DCCM Base Address, AUX_DCCM
0x19	Invalidate Instruction-Cache Line, IC_IVIL
0x1A	Instruction Cache External-Access Address, IC_RAM_ADDR

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name
0x1B	Instruction-Cache Tag Access, IC_TAG
0x1C	Instruction Cache Secure Bit Tag Register, IC_XTAG
0x1D	Instruction Cache Data Access, IC_DATA
0x1E	Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG
0x21	Timer 0 Count Register, COUNT0
0x22	Timer 0 Control Register, CONTROL0
0x23	Timer 0 Limit Register, LIMIT0
0x25	Interrupt Vector Base Register, INT_VECTOR_BASE
0x26	Secure Interrupt Vector Base Register, INT_VECTOR_BASE_S
0x30	Architectural Clock Gating Control Register, ACG_CTRL
0x38	Saved Normal Kernel Stack Pointer, AUX_KERNEL_SP
0x39	Saved Secure User Stack Pointer, AUX_SEC_U_SP
0x3A	Saved Secure Kernel Stack Pointer, AUX_SEC_K_SP
0x3B	Saved Shadow Normal Stack Pointer, AUX_NSEC_SP
0x3F	Error Protection Hardware Control Register, ERP_CTRL
0x40	Register File Error Protection Status Register, RFERP_STATUS_0
0x41	Register File Error Protection Status Register, RFERP_STATUS_1
0x43	Active Interrupts Register, AUX_IRQ_ACT
0x47	Invalidate Data Cache, DC_IVDC
0x48	Data Cache Control Register, DC_CTRL
0x49	Lock Data Cache Line, DC_LDL
0x4A	Invalidate Data Line, DC_IVDL
0x4B	Flush Data Cache, DC_FLSH
0x4C	Flush Data Line, DC_FLDL
0x4D	Data-Cache Region Start Address, DC_STARTR
0x4E	Data-Cache Region End Address, DC_ENDR
0x58	Data Cache External Access Address, DC_RAM_ADDR
0x59	Data Cache Tag Access, DC_TAG
0x5A	Data Cache Secure Bit Register, DC_XTAG
0x5B	Data Cache Data Access, DC_DATA
0x5D	Non-cached Memory Region, AUX_CACHE_LIMIT

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name
0x60 -0x7F	Build Configuration Registers
0xC0 -0xFF	
0x100	Timer 1 Count Register, COUNT1
0x101	Timer 1 Control Register, CONTROL1
0x102	Timer 1 Limit Register, LIMIT1
0x103	RTC Control Register, AUX_RTC_CTRL
0x104	RTC Count Low Register, AUX_RTC_LOW
0x105	RTC Count High Register, AUX_RTC_HIGH
0x106	Secure Timer 0 Count Register, AUX_ST0_COUNT
0x107	Secure Timer 0 Control Register, AUX_ST0_CTRL
0x108	Secure Timer 0 Limit Register, AUX_ST0_LIMIT
0x109	Secure Timer 1 Count Register, AUX_ST1_COUNT
0x10A	Secure Timer 1 Control Register, AUX_ST1_CTRL
0x10B	Secure Timer 1 Limit Register, AUX_ST1_LIMIT
0x200	Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING
0x201	Software Interrupt Trigger, AUX_IRQ_HINT
0x206	Interrupt Priority Register, IRQ_PRIORITY
0x208	ICCM base address, AUX_ICCM
0x20A	Peripheral Memory Region, DMP_PER_AUX
0x20F	ICCM Partition Size Register, CCM_PART_SIZE
0x220	Actionpoint Match Value, AP_AMV0
0x221	Actionpoint Match Mask, AP_AMM0
0x222	Actionpoint Control, AP_AC0
0x223	Actionpoint Match Value, AP_AMV1
0x224	Actionpoint Match Mask, AP_AMM1
0x225	Actionpoint Control, AP_AC1
0x226	Actionpoint Match Value, AP_AMV2
0x227	Actionpoint Match Mask, AP_AMM2
0x228	Actionpoint Control, AP_AC2
0x229	Actionpoint Match Value, AP_AMV3
0x22A	Actionpoint Match Mask, AP_AMM3
0x22B	Actionpoint Control, AP_AC3

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name
0x22C	Actionpoint Match Value, AP_AMV4
0x22D	Actionpoint Match Mask, AP_AMM4
0x22E	Actionpoint Control, AP_AC4
0x22F	Actionpoint Match Value, AP_AMV5
0x230	Actionpoint Match Mask, AP_AMM5
0x231	Actionpoint Control, AP_AC5
0x232	Actionpoint Match Value, AP_AMV6
0x233	Actionpoint Match Mask, AP_AMM6
0x234	Actionpoint Control, AP_AC6
0x235	Actionpoint Match Value, AP_AMV7
0x236	Actionpoint Match Mask, AP_AMM7
0x237	Actionpoint Control, AP_AC7
0x23E	Actionpoint Isolation Enable Register, AP_ISO_EN
0x23F	Watchpoint Program Counter, AP_WP_PC
0x240	Countable Conditions Index Register, CC_INDEX
0x241	Countable Conditions Name0 Register, CC_NAME0
0x242	Countable Conditions Name1 Register, CC_NAME1
0x250	Count-Value Registers, PCT_COUNTL
0x251	Count-Value Registers, PCT_COUNTH
0x252	Snapshot-Value Registers, PCT_SNAPL
0x253	Snapshot-Value Registers, PCT_SNAPH
0x254	Configuration Register, PCT_CONFIG
0x255	Control Register, PCT_CONTROL
0x256	Index-Select Register, PCT_INDEX
0x257	Minimum Value Registers, PCT_MINMAXL
0x258	Maximum Value Registers, PCT_MINMAXH
0x259	Address-Range Registers, PCT_RANGEL
0x25A	Address-Range Registers, PCT_RANGEH
0x25B	User-Flag Register, PCT_UFLAGS
0x260	User Stack Region Top Address, USTACK_TOP
0x261	User Stack Region Base Address, USTACK_BASE
0x262	Secure User Mode Secure Stack Region Top Address, S_STACK_TOP

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name
0x263	Secure User Mode Secure Stack Region Base Address, S_STACK_BASE
0x264	Kernel Stack Region Top Address, KSTACK_TOP
0x265	Kernel Stack Region Base Address, KSTACK_BASE
0x266	Secure Kernel Mode Secure Stack Region Top Address, S_STACK_TOP
0x267	Secure Kernel Mode Secure Stack Region Base Address, S_STACK_BASE
0x268	Secure Jump and Link Indexed Top Address, NSC_TABLE_TOP
0x269	Secure Jump and Link Indexed Base Address, NSC_TABLE_BASE
0x290	Jump and Link Indexed Base Address, JLI_BASE
0x291	Load Indexed Base Address, LDI_BASE
0x292	Execute Indexed Base Address, EI_BASE
0x300	Floating-Point Unit Control Register, FPU_CTRL
0x301	Floating-Point Unit Status Register, FPU_STATUS
0x302	Double-precision Floating Point D1 Lower Register, AUX_DPFP1L
0x303	Double-precision Floating Point D1 Higher Register, AUX_DPFP1H
0x304	Double-precision Floating Point D2 Lower Register, AUX_DPFP2L
0x305	Double-precision Floating Point D2 Higher Register, AUX_DPFP2H
0x350	Key Store Control Register, KEY_STORE_CTRL
0x351	Key Store Password Register, KEY_STORE_PWD
0x352	Key Store Address Register, KEY_STORE_ADDR
0x353	Key Store Data Register, KEY_STORE_DATA
0x354	NV_ICCM Control Register, NV_ICCM_CTRL
0x355	NV_ICCM Erase Register, NV_ICCM_ERASE
0x356	NV_ICCM Programming Register, NV_ICCM_PROG
0x357	NV_ICCM Data Register, NV_ICCM_DATA
0x358	Calibration Parameter Store Control Register, CAL_STORE_CTRL
0x359	Calibration Parameter Store Address Register, CAL_STORE_ADDR
0x35A	Calibration Parameter Store Data Register, CAL_STORE_DATA
0x35C	Key Store SID Register, KEY_STORE_SID
0x380	ARC Trace Address Register, RTT_ADDRESS
0x381	ARC Trace DATA Register, RTT_DATA
0x382	ARC Trace CMD Register, RTT_COMMAND
0x400	Exception Return Address, ERET

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name
0x401	Exception Return Branch Target Address, ERBTA
0x402	Exception Return Status, ERSTATUS
0x403	Exception Cause Register, ECR
0x404	Exception Fault Address, EFA
0x405	ECC Syndrome Register, ESYN
0x406	Exception Secure Status Register, ERSEC_STAT
0x407	Secure Exception Register, AUX_SEC_EXCEPT
0x409	MPU Enable Register, MPU_EN
0x40A	Interrupt Cause Registers, ICAUSE
0x40B	Interrupt Select, IRQ_SELECT
0x40C	Interrupt Enable Register, IRQ_ENABLE
0x40D	Interrupt Trigger Register, IRQ_TRIGGER
0x40F	Interrupt Status Register, IRQ_STATUS
0x410	User Mode Extension Enable Register, XPU
0x412	Branch Target Address, BTA
0x415	Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL
0x416	Interrupt Pending Register, IRQ_PENDING
0x420	MPU Exception Cause Register, MPU_ECR
0x422	MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB15
0x423	MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP15
0x424 to 0x441	Memory Protection Unit Descriptor Base and Permission Registers
0x448	MPU Index Register, MPU_INDEX
0x449	MPU Region Start Address Register, MPU_RSTART
0x44A	MPU Region End Address Register, MPU_REND
0x44B	MPU Region Permission Register, MPU_RPER
0x44C	MPU Probe Register, MPU_PROBE
0x44F	User Extension Flags Register, XFLAGS
0x45D	Start of Translation Region, OLM_VA_START
0x45E	End of Translation Region, OLM_VA_END
0x460	TLB Page Descriptor Register for 32-Bit Virtual Address, TLBPD0
0x461	TLB Page Descriptor Register for 32-Bit Physical Address, TLBPD1
0x464	TLB Index Register, TLBIndex

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name
0x465	TLB Command Register, TLBCommand
0x468	TLB Enable Register, TLBEnable
0x46C	Scratch Data Register, SCRATCH_DATA0
0x490	JLAT Segment 1 Base Address, JOB_BASE1
0x491	JLAT Segment 2 Base Address, JOB_BASE2
0x580	Accumulator Low Register, ACC0_LO
0x581	Low Accumulator Guard and Status Register, ACC0_GLO
0x582	Accumulator High Register, ACC0_HI
0x583	High Accumulator Guard and Status Register, ACC0_GHI
0x598	DSP Butterfly Instructions Data Register, DSP_BFLY0
0x59E	DSP FFT Control Register, DSP_FFT_CTRL
0x59F	DSP Control Register, DSP_CTRL
0x5C0 to 0x5CB	AGU Address Pointer Registers, AGU_AUX_APx
0x5D0 to 0x5D7	AGU Offset Registers, AGU_AUX_OSx
0x5E0 to 0x5F7	AGU Modifier Registers, AGU_AUX_MODx
0x5F8	XCCM Base Address, XCCM_BASE
0x5F9	YCCM Base Address, YCCM_BASE
0x5FC	Bitstream Control Register, BS_AUX_CTRL
0x5FD	Bitstream Base Address Register, BS_AUX_ADDR
0x5FE	Bitstream Offset Register, BS_AUX_BIT_OS
0x5FF	Bitstream Write Data Register, BS_AUX_WDATA
0x600	ARConnect Command Register, CONNECT_CMD
0x601	ARConnect Write Data Register, CONNECT_WDATA
0x602	ARConnect Read Data Register, CONNECT_READBACK
0x610	Core Power Status Register, PDM_PSTAT
0x611	ARC Trace Power Status Register, RTT_PDM_PSTAT
0x612	Core Performance Register, DVFS_PL
0x613	Power Down Register, PDM_PMODE
0x680	DMA Controller Configuration Register, DMACTRL

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name
0x681	DMA Channel Enable Register, DMACENB
0x682	DMA Channel Disable Register, DMACDSB
0x683	DMA Channel High Priority Level Register, DMACHPRI
0x684	DMA Channel Normal Priority Level Register, DMACNPRI
0x685	DMA Channel Transfer Request Register, DMACREQ
0x686	DMA Channel Status Register, DMACSTAT0
0x687	DMA Channel Status Register, DMACSTAT1
0x688	DMA Channel Interrupt Request Status Register, DMACIRQ
0x689	DMA Channel Structure Base Address Register, DMACBASE
0x68A	DMA Channel Reset Register, DMACRST
See Table 44-12, Table 44-13, Table 44-14	DMA Channel Control Register, DMACTRLx
See Table 44-12, Table 44-13, Table 44-14	DMA Channel Source Address Register, DMASARx
See Table 44-12, Table 44-13, Table 44-14	DMA Channel Destination Address Register, DMADARx
Memory-mapped	DMA Channel Linked-List Pointer Register, DMALLPx
0x700	SmaRT Control Register, SMART_CONTROL
0x701	SmaRT Data Register, SMART_DATA
0xE00	Watchdog Password Register, EWDT_PASSWD
0xE01	Watchdog Time Error Status Register, EWDT_ESR
0xE02	Watchdog Timer Index Register, EWDT_INDEX
0xE03	Watchdog Control Register, EWDT_CTRLx
0xE04	Watchdog Timer Period Register, EWDT_PERIODx
0xE05	Watchdog Timer Counter Register, EWDT_COUNTx
0xF000	Key Data Register, AUX_KEY_DATA
0xF001	Key Unlock Register, AUX_KEY_UNLOCK

1.1 Build Configuration Registers

The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

A set of baseline Build Configuration Registers are present in all ARCV2 processors. In addition, each optional ISA extension typically includes a Build Configuration Register to signify its presence and its specific configuration in each given build.

Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCV2-based system.

Auxiliary registers, in the range 0x60 to 0x7F and 0xC0 to 0xFF, are assumed to be BCRs. In kernel mode, any read from a non-existent build configuration register in this range returns 0, and no exception is generated. This design enables the kernel-mode code to detect the presence or absence of a BCR because all BCRs that are present in a system contain non-zero values.

However, in user mode, reads from build configuration registers always raise a Privilege Violation exception (see [Privilege Violation, Kernel/User Only Access](#)).

Any write to a build configuration register, whether in kernel or user mode, raise an [Illegal Instruction](#) exception.

[Table 1-2](#) summarizes the build configuration registers for components that are described in this document.

Table 1-2 Build Configuration Registers

Number	Name
0x60	Build Configuration Registers Version, BCR_VER
0x63	BTA Configuration Register, BTA_LINK_BUILD
0x68	Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD
0x6D	Memory Protection Build Configuration Register, MPU_BUILD
0x6E	Core Register File Configuration Register, RF_BUILD
0x6F	OLM Build Configuration Register, OLM_BUILD
0x70	Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD
0x72	Data Cache Configuration Register, D_CACHE_BUILD
0x74	DCCM Configuration Register, DCCM_BUILD
0x75	Timers Configuration Register, TIMER_BUILD
0x76	Actionpoints Configuration Register, AP_BUILD
0x77	Instruction Cache Configuration Register, I_CACHE_BUILD
0x78	ICCM Configuration Register, ICCM_BUILD

Table 1-2 Build Configuration Registers (Continued)

Number	Name
0x79	XY Build Configuration Register, XY_BUILD
0x7A	DSP Build Configuration Register, DSP_BUILD
0x7B	Multiplier Configuration Register, MULTIPLY_BUILD
0x7C	Swap Instruction Configuration Register, SWAP_BUILD
0x7D	Normalize Instruction Configuration Register, NORM_BUILD
0x7E	Min/Max Instruction Configuration Register, MINMAX_BUILD
0x7F	Barrel Shifter Configuration Register, BARREL_BUILD
0xC1	Instruction Set Configuration Register, ISA_CONFIG
0xC3	Peripheral Region Build Configuration Register, DMP_PP_BUILD
0xC5	Stack Region Configuration Register, STACK_REGION_BUILD
0xC7	Error Protection Configuration Register, ERP_BUILD
0xC8	Floating-Point Unit Build Register, FPU_BUILD
0xC9	Code Protection Register, CPROT_BUILD
0xCB	Bitstream Identity Register, BS_BUILD
0xCC	AGU Build Configuration Register, AGU_BUILD
0xCD	DMA Build Configuration Register, DMAC_BUILD
0xD0	ARConnect Build Configuration Register, CONNECT_SYSTEM_BUILD
0xD1	Inter-core Semaphore Unit BCR, CONNECT_SEMA_BUILD
0xD2	Inter-core Message Unit BCR, CONNECT_MESSAGE_BUILD
0xD3	Power Management Unit BCR, CONNECT_PMU_BUILD
0xD6	ARConnect Global Free Running Counter BCR, CONNECT_GFRC_BUILD
0xD8	Key Store Build Configuration Register, KEY_STORE_BUILD
0xD9	Calibration Parameter Store Build Configuration Register, CAL_STORE_BUILD
0xDB	Secure Build Configuration Register, SEC_BUILD
0xE0	Inter-Core Interrupt Unit BCR, CONNECT_ICI_BUILD
0xE1	Inter-Core Debug Unit BCR, CONNECT_ICD_BUILD
0xE3	ARConnect PDM BCR, CONNECT_PDM_BUILD

Table 1-2 Build Configuration Registers (Continued)

Number	Name
0xEE	Watchdog Timer Build Configuration Register, EWDT_BUILD
0xF1	Core Architecture Build Configuration Register, ARCH_CONFIG
0xF2	ARC Trace Build Configuration Register, RTT_BUILD
0xF3	Interrupt Build Configuration Register, IRQ_BUILD
0xF5	Performance Counter Build-Configuration Register, PCT_BUILD
0xF6	Countable Conditions Build Configuration Register, CC_BUILD
0xF7	Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD
0xFE	Instruction Fetch Queue Configuration Register, IFQUEUE_BUILD
0xFF	SMART_BUILD Configuration Register, SMART_BUILD
0xF80	Task Switch Build Register, TASK_SWITCH_BUILD

1.1.1 Serializing Auxiliary-Register Writes

Writes to certain auxiliary registers must take effect from the instruction immediately after the SR instruction that updates those registers. This is enforced in the processor by designating such auxiliary registers as *serializing*, and ensuring that a pipeline flush takes place immediately after any SR to a serializing register commits.

The base set of serializing auxiliary registers is:

- LP_START
- LP_END
- JLI_BASE
- All actionpoint auxiliary registers
- All auxiliary registers related to data cache
- All auxiliary registers related to instruction cache
- All auxiliary registers related to MPU

The pipeline flush operation requires that all pending state update transactions are completed prior to the serializing operation taking effect. The cycle penalty would vary, based on pending post-commit instructions or buffered stores to memory. Pipeline restart takes additional 10 clocks until the next instruction commits. Serializing writes take place very infrequently in most applications.

Extension auxiliary registers may also be designated as serializing; see the *APEX Databook* for more information. Similarly, external user-defined auxiliary registers (UAUX) may force serialization.

2

Exceptions List



Note

Table 2-1 indicates the ECR values when an exception is raised. If simultaneous violations occur, the Parameter field in ECR has one bit set for each of those exceptions. For example, if there are simultaneous Code Protection, Stack Checking memory write violations, the Parameter field reads 3 (bit 0 is set for code protection and bit 1 is set for stack checking).

Table 2-1 Exception Vectors and Cause Codes

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Reset	Reset	0x00	0x00	0x00	0x00	0x000000	Always
Bus Error from Instruction Memory	Memory Error	0x04	0x01	0x00	0x0u	0x01000u	Always
Bus Error from Data Memory	Memory Error	0x04	0x01	0x10	0x0u	0x01100u'(b)	
Instruction Fetch spanning multiple instruction memory targets	Memory Error	0x04	0x01	0x02	0x00	0x010200	
Instruction Fetch spanning multiple MPU regions	Memory Error	0x04	0x01	0x03	0x00	0x010300	Always
Data access spanning multiple data memory targets	Memory Error	0x04	0x01	0x12	0x00	0x011200	
NVM write or erase programming failure	Memory Error	0x04	0x01	0x13	0x00	0x011300	

Table 2-1 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
NVM address out of range (instruction fetch or data access) only when NVM is configured	Memory Error	0x04	0x01	0x14	0x00	0x011400	
XY access to illegal target memory	Memory Error	0x04	0x01	0x21	0x00	0x012100	
XY access spanning multiple targets	Memory Error	0x04	0x01	0x22	0x00	0x012200	
XY 64-bit destination is not 32-bit aligned	Memory Error	0x04	0x01	0x23	0x00	0x012300	
Illegal Instruction	Instruction Error	0x08	0x02	0x00	0x00	0x020000	
Illegal Instruction Sequence	Instruction Error	0x08	0x02	0x01	0x00	0x020100	
Illegal instruction execution from S to NS (branch w/o link)	Instruction Error	0x08	0x02	0x10	0x00	0x021000	
Illegal instruction sequence, linear execution from S to NS	Instruction Error	0x08	0x02	0x11	0x00	0x021100	
Illegal instruction execution from NS to S (non-linear)	Instruction Error	0x08	0x02	0x12	0x00	0x021200	
Illegal instruction sequence, linear execution from NS to S	Instruction Error	0x08	0x02	0x13	0x00	0x021300	
Double Fault	EV_MachineCheck	0x0C	0x03	0x00	0x00	0x030000	
Overlapping TLB Entries	EV_MachineCheck	0x0C	0x03	0x01	0x00	0x030100	
Fatal TLB Error	EV_MachineCheck	0x0C	0x03	0x02	0x00	0x0302nn(h)	
Internal Memory Error on Instruction Fetch	EV_MachineCheck	0x0C	0x03	0x04	0xrr	0x0304rr(e)	
Internal Memory Error on Data Access	EV_MachineCheck	0x0C	0x03	0x05	0xrr	0x0305rr(e)	

Table 2-1 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Illegal Overlapping MPU Entries	EV_MachineCheck	0x0C	0x03	0x06	0x00	0x030600	Always
Illegal Overlapping MPU entries (jump and branch target)	EV_MachineCheck	0x0C	0x03	0x06	0x01	0x030601(f)	Always
Secure Vector Table not located in Secure memory	EV_MachineCheck	0x0C	0x03	0x10	0x00	0x031000	Always
NSC jump table not located in Secure memory	EV_MachineCheck	0x0C	0x03	0x11	0x00	0x031100(f)	Always
Secure handler code not located in secure memory	EV_MachineCheck	0x0C	0x03	0x12	0x00	0x031200	Always
NSC target address not located in Secure memory	EV_MachineCheck	0x0C	0x03	0x13	0x00	0x031300	Always
Non-Maskable imprecise exception	EV_MachineCheck	0x0C	0x03	0x14	0x00	0x031400	
Non-maskable imprecise exception double fault	EV_MachineCheck	0x0C	0x03	0x14	0x01	0x031401	
Instruction Fetch TLB Miss	EV_ITLBMiss	0x10	0x04	0x00	0x00	0x040000	
Data TLB Miss on read	EV_DTLBMiss	0x14	0x05	0x01	0x00	0x050100	
Data TLB miss on write	EV_DTLBMiss	0x14	0x05	0x02	0x00	0x050200	
Data TLB miss on RMW	EV_DTLBMiss	0x14	0x05	0x03	0x00	0x050300	
Instruction Fetch Protection Violation in MPU	EV_ProtV	0x18	0x06	0x00	0x04	0x060004	
Instruction Fetch Protection Violation in OLM(d)	EV_ProtV	0x18	0x06	0x00	0x08	0x060008	
Instruction Fetch Protection Violation on invalid Secure/Normal transition	EV_ProtV	0x18	0x06	0x00	0x24	0x060024	Always
Instruction-fetch invalid SID value (multiple bits set)	EV_ProtV	0x18	0x06	0x00	0x44	0x060044	Always

Table 2-1 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in code protection scheme (parameter code 0x01)	EV_ProtV	0x18	0x06	0x01	0x01	0x060101	
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in stack checking scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x01	0x02	0x060102	
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in MPU (parameter code 0x04)	EV_ProtV	0x18	0x06	0x01	0x04	0x060104	
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in OLM (parameter code 0x08)(d)	EV_ProtV	0x18	0x06	0x01	0x08	0x060108	
Memory Read Protection Violation from Secure MPU	EV_ProtV	0x18	0x06	0x01	0x24	0x060124	Always
Memory Read Protection Violation from Secure MPU with SID Mismatch	EV_ProtV	0x18	0x06	0x01	0x44	0x060144	Always
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in code protection scheme (parameter code 0x01)	EV_ProtV	0x18	0x06	0x02	0x01	0x060201	
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in stack checking scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x02	0x02	0x060202	
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in MPU (parameter code 0x04)	EV_ProtV	0x18	0x06	0x02	0x04	0x060204	

Table 2-1 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in OLM (parameter code 0x08)(d)	EV_ProtV	0x18	0x06	0x02	0x08	0x060208	
Memory Write Protection Violation from NVM	EV_ProtV	0x18	0x06	0x02	0x10	0x060210	
Memory Write Protection Violation from Secure MPU	EV_ProtV	0x18	0x06	0x02	0x24	0x060224	Always
Memory Write Protection Violation from Secure MPU with SID Mismatch	EV_ProtV	0x18	0x06	0x02	0x44	0x060244	Always
Memory Read-Modify-Write (EX) protection violation in code protection (parameter code 0x01)	EV_ProtV	0x18	0x06	0x03	0x01	0x060301	
Memory Read-Modify-Write (EX) protection violation in stack checking protection scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x03	0x02	0x060302	
Memory Read-Modify-Write (EX) protection violation in MPU scheme (parameter code 0x04)	EV_ProtV	0x18	0x06	0x03	0x04	0x060304	
Memory Read-Modify-Write (EX) protection violation in OLM scheme (parameter code 0x08)(d)	EV_ProtV	0x18	0x06	0x03	0x08	0x060308	
Memory Read-Modify-Write Protection Violation from NVM	EV_ProtV	0x18	0x06	0x03	0x10	0x060310	
Memory Read-Modify-Write Protection Violation from Secure MPU	EV_ProtV	0x18	0x06	0x03	0x24	0x060324	Always

Table 2-1 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Memory Read-Modify-Write Protection Violation from Secure MPU with SID Mismatch	EV_ProtV	0x18	0x06	0x03	0x44	0x060344	Always
Normal Vector Table in Secure Memory	EV_ProtV	0x18	0x06	0x10	0x00	0x061000	Always
NS handler code located in S memory	EV_ProtV	0x18	0x06	0x11	0x00	0x061100	Always
NSC Table Range Violation	EV_ProtV	0x18	0x06	0x12	0x00	0x061200	Always
Action Point Hit, Instruction Fetch	EV_PrivilegeV	0x1C	0x07	0x02	0xnn	0x0702nn	
Privilege Violation	EV_PrivilegeV	0x1C	0x07	0x00	0x00	0x070000	
Disabled Extension	EV_PrivilegeV	0x1C	0x07	0x01	0xnn	0x0701nn	
Action Point Hit, Memory or Register	EV_PrivilegeV	0x1C	0x07	0x02	0xnn	0x0702nn	
N to S Return Using Incorrect Return Mechanism	EV_PrivilegeV	0x1C	0x07	0x10	0x01	0x071001	Always
N to S Return with Incorrect Operating Mode	EV_PrivilegeV	0x1C	0x07	0x10	0x02	0x071002	Always
IRQ/Exception Return fetch from wrong mode	EV_PrivilegeV	0x1C	0x07	0x10	0x03	0x071003	Always
Attempt to halt secure processor in NS mode(g)	EV_PrivilegeV	0x1C	0x07	0x10	0x04	0x071004	Always
Attempt to access secure resource from normal mode	EV_PrivilegeV	0x1C	0x07	0x10	0x20	0x071020	Always
SID Violation on Resource Access (APEX / UAUX / key NVM)	EV_PrivilegeV	0x1C	0x07	0x10	0x40	0x071040	Always
Attempt to access Secure APEX feature from NS mode	EV_PrivilegeV	0x1C	0x07	0x13	0x20	0x071320	Always
SID violation on access to APEX feature	EV_PrivilegeV	0x1C	0x07	0x13	0x40	0x071340	Always
Software Interrupt	EV_SWI	0x20	0x08	0x00	0xnn	0x0800nn	

Table 2-1 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Trap	EV_Trap	0x24	0x09	0x00	0xnn	0x0900nn	
Extension Instruction Exception	EV_Extension	0x28	0x0A	mm	0xnn	0x0Ammnn(c)	
Invalid Operation, Floating-point extension exceptions	EV_Extension	0x28	0x0A	0x00	0x01	0x0A0001	
Divide by Zero, Floating-point extension exceptions	EV_Extension	0x28	0x0A	0x00	0x02	0x0A0002	
Divide by zero, Integer exception	EV_DivZero	0x2C	0x0B	0x00	0x00	0x0B0000	
Misaligned data access	EV_Misaligned	0x34	0x0D	0x00	0x00	0x0D0000	
Reserved	-	0x3C	0x0F	-		-	

- (a) The value of 'u' in the parameter field of Bus Error exceptions is determined by the value of STATUS32[U] bit at the time that the instruction triggering the exception was committed. Hence, a User-mode instruction fetch that results in a bus error from external memory, raises an exception with ECR 0x010001. The same exception in Kernel mode has an ECR value of 0x010000.
- (b) A load/store access targeting ICCM may be treated in some implementations as an external memory access (external to the Data Memory interface); in this case, if a memory error such as uncorrectable ECC error is encountered in either a load or partial store (RMW) to ICCM, a Bus Error from Data Memory exception may be raised. See [“Bus Error on Data Access to External Data Memory”](#) on page 286.
- (c) An example of an extension is floating-point extension. For this extension, the following are the values of sub cause code and parameter:

mm = 0 – indicates floating-point extension exceptions

nn—indicates specific exception(s) raised

nn	=	1 – Invalid Operation floating-point exception
	=	2 – Divide by Zero floating-point exception

- (d) The rules for prioritizing exceptions are applied normally, when multiple MMU exceptions are raised on two adjacent pages accessed by non-aligned memory operations. That means that the highest priority exceptions are always taken in preference to lower priority exceptions. If two exceptions of the same type occur, for different pages, from the same memory reference, then the exception triggered by the effective byte address (EBA) of the memory operation should be taken first. If a reference spans two pages, then the second address will be EBA+4.
- (e) The parameter codes (rr) are implementation dependent. See the processor databook for more information about the error parameter codes for this exception.
- (f) For SJLI instruction, this exception occurs after the SJLI instruction commits.
- (g) BRK/BRK_S instructions can halt the processor in the normal kernel mode. In that case, exception is not raised.
- (h) Fatal TLB error Parameter code:
- nn==04 when ECC or parity errors occur in instruction cache data during instruction fetch
 - nn==05 when ECC or parity errors occur in instruction cache tag memory during instruction fetch
 - nn==06 when ECC errors occur in address during instruction fetch

Part 1

ARCV2 Baseline ISA

In this part:

- [Introduction](#)
- [Data Organization and Addressing](#)
- [Core Architectural State Registers](#)
- [Interrupts and Exceptions](#)
- [Instruction Set Summary](#)
- [32-bit Instruction Formats Reference](#)
- [16-bit Instruction Formats Reference](#)
- [Instruction Set Details](#)

3

Introduction

This document is intended for programmers of the ARCV2 ISA (Instruction-Set Architecture). The ARCV2 ISA comprises a mandatory set of baseline features, together with a collection of optional extensions to the ISA. This document covers all aspects of the ARCV2 ISA.

The ARCV2 ISA is designed to provide highly compact encodings, while offering high performance. There is also a significant amount of opcode space available for extension instructions. In the ARCV2 ISA, compact 16-bit encodings of frequently used 32-bit instructions are provided. These can be freely intermixed with 32-bit encodings.

3.1 Typographic Conventions

- Normal text is displayed using this font.
- Code segments are displayed in this mono-space font.
- Deprecated instruction mnemonics are grayed, for example:

LD LDH LDW LDB



Note

Notes point out important information.



Caution

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

3.2 Key Features of the ARCV2 ISA

The following are the main features of ARCV2 ISA:

- Instructions
 - Freely intermixed 16 and 32-bit instruction formats
 - User and kernel modes

- Registers
 - General Purpose Core Registers
 - Special Purpose Auxiliary Register Set
- Memory Addressing Modes
 - Address Register Write-Back
 - Pre- and Post-Address Register Write-Back
 - Stack Pointer Support
 - Scaled Data Size Addressing Mode
 - PC-relative Addressing
- Program Flow
 - Conditional ALU Instructions
 - Single Cycle Immediate Data
 - Jumps and Branches with Single Instruction Delay Slot
 - Combined compare-and-branch instructions
 - Delay Slot Execution Modes
 - Zero Overhead Loops
- Interrupts and Exceptions
 - Dynamically Assignable Interrupt Priorities
 - Non-Maskable Exceptions
 - Maskable External Interrupts
 - Precise Exceptions
 - Memory, Instruction, and Privilege Exceptions
 - Exception Recovery State
 - Exception Vectors
 - Exception Return Instruction
- Extension Packs
 - Code Density options
 - Bit manipulation and normalization
 - Byte-level shifts, rotates, and endianness reordering
 - Low-cost byte and half-word (16-bits) shifts and rotations
 - Integer multiplication options
 - Integer division
 - Synchronization
- User-defined APEX extensions

- ❑ User-defined instructions
- ❑ User-defined predication conditions
- ❑ User-defined auxiliary registers
- ❑ Up to 26 additional extension core registers
- Floating-point Unit
 - ❑ Single-precision floating-point instructions
 - ❑ Double-precision floating-point instructions
- Power Management
 - ❑ Sleep Instruction
 - ❑

3.3 Programmer's Model

The programmer's model is common to all implementations of the ARCV2-based processors and allows upward compatibility of code for software that has been compiled for similarly-configured ARCV2 processors.

3.3.1 Core Register Set

ARCV2-based processors provide a set of general-purpose registers, allowing instructions to have up to two source operands and one destination. Programmer may use the general purpose registers (r0-r25, and r30) for any purpose. Some of the core registers have a defined purpose such as stack pointers, link registers, and loop counters. See section [“Core Register Set”](#) on page 131.

Additional banks of registers may be configured for fast interrupt response and context switching. See [“Banked Interrupt Registers”](#) on page 237.

3.3.2 Auxiliary Register Set

The auxiliary register set contains status and control registers, which by default are 32 bits wide. These auxiliary registers occupy a separate 32-bit address space from the normal memory-access (that is, load and store) instructions. Auxiliary registers accessed using distinct Load Register (LR), Store Register (SR), and Auxiliary EXchange (AEX) instructions. See [“Auxiliary Register Set”](#) on page 141.

3.3.3 32-bit Instruction Formats

The majority of instructions in the ARCV2 ISA can be encoded in a 32-bit format. This instruction format provides access to the full set of registers and operational variants of each instruction. Dyadic register-to-register instructions typically provide three independent register operands, that is, two source registers and one destination register. Alternatively, the second source operand can be specified as a short immediate value, contained within each 32-bit instruction.

When a full 32-bit long immediate operand (see [Immediate Data Indicator, L IMM \(r62\)](#)) is required, the requirement is indicated by specifying register r62 as a source register. This instruction requires the 32-bit literal operand to be present in the next consecutive four bytes of memory, after the instruction.

Register r63 (see [Word-aligned Program Counter, PCL \(r63\)](#)) is a read-only register containing the 32-bit word-aligned PC (see [Program Counter, PC](#)) for use as a source operand in all instructions, thereby supporting PC-relative addressing. Bits 0 and 1 of PCL are always read as zero.

3.3.4 16-bit Instruction Formats

There are also a range of compact 16-bit encodings of frequently occurring instructions, and a selection of instructions encoded in 16-bit formats which do not require the full 32-bit format. The 16-bit instruction format offers the following:

- A reduced set of register operands, limited to the following most frequent eight registers: r0-r3, r12-r15
- The use of implied registers, such as BLINK (see [Link Registers, ILINK \(r29\), BLINK \(r31\)](#)), SP, GP, FP (see [Pointer Registers, GP \(r26\), FP \(r27\), SP \(r28\)](#)), and PC (see [Program Counter, PC](#)).
- A reduced number of distinct operands, limited to 1 or 2 operand registers, by sharing the destination register with one of the source registers
- Reduced immediate data sizes
- Reduced branch range from maximum offset of $\pm 16\text{MB}$ to maximum offset of $\pm 512\text{B}$
- No branch delay slot execution modes
- No conditional execution
- No flag setting option (only few instructions set flags, for example, BTST_S (see [BTST](#)), CMP_S (see [CMP](#)), and TST_S (see [TST](#)))

3.3.5 Operating Privileges

The ARCV2 architecture supports two distinct operating privilege levels to allow different levels of privilege to be assigned to operating system kernels and user programs. These operating modes, along with the memory management and protection features, ensure the following:

- An OS can maintain control of the system at all times.
- The OS and user tasks can be protected from a malfunctioning or malicious user task.
- A user task cannot amplify its own privileges.

The operating mode is used to determine whether a privileged instruction may be executed or a privileged register can be accessed. The operating mode is also used by the memory management system to determine whether a specific location in memory may be accessed. The ARCV2 architecture supports the following two distinct privilege modes:

- Kernel mode: Provides the following:
 - Highest level of privilege
 - Default mode from [Reset](#)
 - Access to all machine state, including privileged instructions and privileged registers
- User mode: Provides the following:
 - Lowest level of privilege

- ❑ Limited access to machine state
- ❑ Any attempt to access privileged machine state raises an exception

When the option `-sec_modes_option == true`, secure and normal operating modes are available that are orthogonal to the kernel and user modes. This allows the processor to support four distinct operating privilege levels: two baseline operating levels: the normal user and kernel mode, secure user and kernel operating levels. For more information, see [Operating Privileges](#).

3.4 Configurability

The ARCV2 architecture offers several well-defined methods for configuring the programming model and instruction set. In addition, each processor within the ARCV2 family offers well-defined methods for configuring additional core components, and to select from the available micro-architectural implementation options to choose an appropriate trade-off between performance, complexity, operating frequency, and energy consumption. This section defines the full set of options for configuring the programming model, instruction set architecture, and the additional core components. Micro-architectural configuration options are specific to each ARCV2 implementation, and are therefore described in the relevant *Databook* for each processor.

[Table 3-1](#) describes the configuration options for the programming model supported by the ARCV2 architecture.



Note

Each individual processor may support a subset of these options. See to the relevant *Databook* for each processor.

3.4.1 Programming Model Options

Various aspects of the programming model of an ARCV2 processor can be configured. These include the byte ordering of memory, the number of implemented bits in each address or program counter value, the configuration of the general-purpose register file, and the configuration of the interrupt mechanism. The full set of programming model options is listed in [Table 3-1](#).

Table 3-1 ARCV2 Programming Model Configuration Options

Configuration Option	Value Range	Description
<code>-byte_order</code>	little, big	Defines the ordering of bytes within a multi-byte memory word.
<code>-addr_size</code>	16, 20, 24, 28, 32 bits	Determines the address-bus width throughout the core. For processors with virtual memory support, this option defines the number of bits in a virtual address.
<code>-pc_size</code>	16, 20, 24, 28, 32 bits	Determines the maximum addressable code space. This option impacts PC-related hardware. This value cannot exceed <code>-addr_size</code> , but can be smaller than or the same as <code>-addr_size</code> .

Table 3-1 ARCV2 Programming Model Configuration Options (Continued)

Configuration Option	Value Range	Description
-lpc_size	0, 8, 12, 16, 20, 24, 28 or 32 bits	Defines the number of bits in the LP_COUNT register.
-rgf_num_regs	16, 32 registers	Defines the number of registers in the primary register bank supported by the processor. If 16 registers are selected, only registers in the range R0-R3, R10-R15, and R26-R31 are present.
-rgf_banked_regs	4, 8, 16, 32	Defines the number of registers replicated per register bank.
-rgf_num_banks	1 through 8	Defines the number of register file banks. Register bank 0 to Register bank 7. The ARCV2ISA supports up to 8 register banks. However, the processors might not implement all 8 register banks. See the processor databook for the number of register banks implemented in that processor.
-intvbase_preset	PC range	(INTVBASE_PRESET x 1024) Defines the reset value of the programmable INT_VECTOR_BASE register. This parameter is also reflected in the ADDR field of the read only VECBASE_AC_BUILD register. On reset, program execution begins at the address referenced by the reset vector fetched from the 1 KB aligned address (INTVBASE_PRESET x 1024). This option is used for the normal interrupts.
-has_interrupts	false, true	Indicates if the processor configuration includes the interrupt unit. ■ false: indicates that the interrupt unit is not included. ■ true: indicates that the interrupt unit is included.
-halt_on_reset	false, true	Indicates whether the core is halted initially on reset. ■ false: core is not halted initially on reset. ■ true: core is halted initially on reset.
-number_of_interrupts	0 to 240 interrupts	Defines the number of interrupts in the interrupt controller. Note: This option is available only if -has_interrupt==true.
-number_of_levels	1 to 16	Defines the number of interrupt priority levels (0 to 15) in the interrupt controller. Note: This option is available only if -has_interrupt==true.
-firq_option	false, true	Indicates whether the fast interrupt option is enabled. ■ false: indicates that the fast interrupt option is disabled. ■ true: indicates that the fast interrupt option is enabled. Note: This option is available only if -has_interrupt==true.

Table 3-1 ARCV2 Programming Model Configuration Options (Continued)

Configuration Option	Value Range	Description
<code>-nmi_option</code>	0, 1, 2	Indicates whether the core includes an external signal using which you can trigger a non-maskable imprecise exception. ■ 0: not supported ■ 1: includes the non-maskable signal for a single-core ■ 2: includes the non-maskable signal for multicore

3.4.2 Processor Component Options

ARCV2 processors may include a number of additional components that extend the capabilities of the processor in areas such as memory management, debugging, timers, and so on. These components extend the programming model of the processor through the provision of additional auxiliary registers, and through the addition of extra conditions under which exceptions or interrupts may be generated.

For a list of the processor component options, see the ARCV2-based processor databook.

3.4.3 ISA Options

[Table 3-2](#) summarizes the set of supported instruction set options in the ARCV2 architecture.

Column 1 lists the ISA configuration options. Column 2 indicates the set of permissible values for each configuration option, and columns 3 and 4 describe the instructions and auxiliary registers added to the architecture in each case. Each of these instructions is described in [“Instruction Set Details”](#) on page 409.

Table 3-2 ARCV2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-code_density_option	false	-	-
	true	SETEQ, SETNE, SETLT, SETGE, SETLO, SETHS, SETLE, SETGT, ENTER_S, LEAVE_S, BI, BIH, LDI, LDI_S, LD_S R0-3,[h,u5] LD_S. AS a,[b,c] LD_S R1, [GP, s11] LDH_S, LDB_S, ST_S R0,[GP,s11], STH_S, STB_S, JLI_S, EI_S,	JLI_BASE, LDI_BASE, EI_BASE
-bitscan_option	false	-	-
	true	NORM, NORMH, FFS, FLS	-
-swap_option	false	-	-
	true	SWAP, SWAPE, LSL16, LSR16	-

Table 3-2 ARCV2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-shift_option	0	-	-
	1	ASR16, ASR8, LSR8, LSL8, ROL8, ROR8	
	2	Multi-bit shift or rotate operations: ASL, LSR, ASR, ROR, ASL_S, LSR_S, ASR_S. XBFU (when -bitfield_option ==true).	-
	3	ASR16, ASR8, LSR8, LSL8, ROL8, ROR8 Multi-bit shift or rotate operations: ASL, LSR, ASR, ROR, ASL_S, LSR_S, ASR_S. XBFU (when -bitfield_option ==true).	
-bitfield_option	false	-	
	true	XBFU (when SHIFT_OPTION is 2 or 3). Note: this option is enabled by default for cores with ARCOVER value of 0x50 or above.	
-os_opt_option	false	-	
	true	PREALLOC, SWI_S encoding with n6 operand, DSYNC, DMB	
-mpy_option	none	-	-
	wlh1	MPYW, MPYUW	-
	wlh2 to wlh5(also referred to as mpy_option==7 in some places in the document)	All mpy_option=wlh1 instructions and MPY, MPYU, MPYM, MPYMU, MPY_S,	-

Table 3-2 ARCV2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-atomic_option	false		-
	true	LLOCK, SCND, WLFC LLOCKD and SCND instructions also require -ll64_option==true	-
-div_rem_option	none	-	DZ bit in STATUS32
	radix4_enhanced	DIV, DIVU, REM, REMU	
-has_fpu	true	FSMUL, FSADD, FSSUB, FSCMP, FSCMPF, FS2INT, FS2INT_RZ, FINT2S, FS2UINT, FS2UINT_RZ, FUINT2S, FSABS, FSNEG,	
-fpu_div_option	false	-	
	true	if -fpu_dp_option ==false: FSDIV, FSSQRT if -fpu_dp_option ==true: FSDIV, FSSQRT, FDDIV, FDSQRT	
-fpu_fma_option	false	-	
	true	if -fpu_dp_option ==false: FSMADD, FSMSUB if -fpu_dp_option ==true: FSMADD, FSMSUB, FDMADD, FDMSUB	ACCH, ACCL
-fpu_dp_assist	true	DMULH11, DMULH12, DMULH21, DMULH22, DADDH11, DADDH12, DADDH21, DADDH22, DSUBH11, DSUBH12, DSUBH21, DSUBH22, DEXCL1, DEXCL2	
-has_dsp	true	See Table 48-1	
-dsp_complex	true	Instructions included when has_dsp==true and instructions listed in Table 7-53 on page 354.	

Table 3-2 **ARCV2 Instruction Set Options**

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-sec_modes_option	false	-	
	true	SJLI, SFLAG	■ NSC_TABLE_TOP ■ NSC_TABLE_BASE ■ AUX_KERNEL_SP ■ AUX_SEC_U_SP ■ AUX_SEC_K_SP ■ AUX_SEC_EXCEPT ■ ERSEC_STAT ■ SEC_STAT ■ SEC_VECBASE_BUILD ■ Note: The -sec_modes_option is not a baseline feature. This feature is available with the SecureShield component. However, this option affects the core instruction and architectural state. For more information about this option, see SecureShield™ Technology.
-intvbase_preset_s	PC range		INT_VECTOR_BASE_S register. (INT_VECTOR_BASE_S x 1024) Defines the reset value of the programmable INT_VECTOR_BASE_S register. This parameter is also reflected in the ADDR field of the read only SEC_VECBASE_BUILD register. On reset, program execution begins at the address referenced by the reset vector fetched from the 1 KB aligned address (INT_VECTOR_BASE_S x 1024). This option is used for configuring the secure interrupts. Note: This option is available only when – sec_modes_option==true.

3.5 Custom Extensions

The ARCV2-based processor is designed to be extensible according to the requirements of the system in which the processor is used. These extensions may include more core and auxiliary registers, new instructions, and additional condition code tests. This section provides information about where processor extensions occur and how they affect the programmer's view of the ARCV2-based processor.

3.5.1 Extension Core Registers

The core register set has a total of 64 different addressable registers. Registers `r32` to `r57` are available for extension purposes. [Figure 5-1](#) on page 131 shows the core register map.

3.5.2 Extension Auxiliary Registers

The auxiliary registers are accessed with 32-bit addresses and are of 32-bit word data size. Except for the positions defined as basecase for auxiliary registers, the extensions to the auxiliary register set can be anywhere in this address space. These are referred to by using the load from auxiliary register ([LR](#)) and store to auxiliary register ([SR](#)) instructions or special extension instructions.

The auxiliary register address region `0x60` up to `0x7F` and region `0xC0` up to `0xFF` is reserved for the [Build Configuration Registers](#) (BCRs) that can be used by embedded software or host debug software to detect the configuration of the ARCV2-based hardware. The Build Configuration Registers contain the version of each ARCV2-based extension and also the build-specific configuration information.

Some optional components in an ARCV2-based processor system may provide only version information registers to indicate the presence of a given component. These *version registers* are not necessarily part of the Build Configuration Registers set. These optional component version registers may be provided as part of the extension auxiliary register set for a component.

3.5.3 Extension Instructions

Instruction groups are encoded within the instruction word using a 5-bit binary field. The first 8 encodings define 32-bit instruction groups; the remaining 24 encodings define 16-bit instruction groups.

The following two extension instruction groups are provided in the 32-bit instruction set:

- 1 reserved extension group
- 1 user extension group

The reserved extension group and the user extension group can contain dual-operand instructions (**a b op c**), single-operand instructions (**a op b**), and zero-operand instructions (**op c**). These types of extension instructions are used in the same way as the normal ALU instructions, except an external ALU is used to obtain the result for write-back to the core register set.

3.5.4 Extension Condition Codes

The condition code test on an instruction is encoded by using a 5-bit binary field which gives 32 different possible conditions that can be tested. The first 16 codes (`0x00-0x0F`) are the condition codes defined in the basecase version of ARCV2-based processor which use only the internal condition flags from the status register (`Z`, `N`, `C`, `V`). For more information, see [Table 7-7](#) on page 308.

The remaining 16 condition codes (`10-1F`) are available for extension and are used to do the following:

- Provide additional tests on the internal condition flags
- Test extension status flags from extension function units
- Test a combination of external and internal flags

4

Data Organization and Addressing

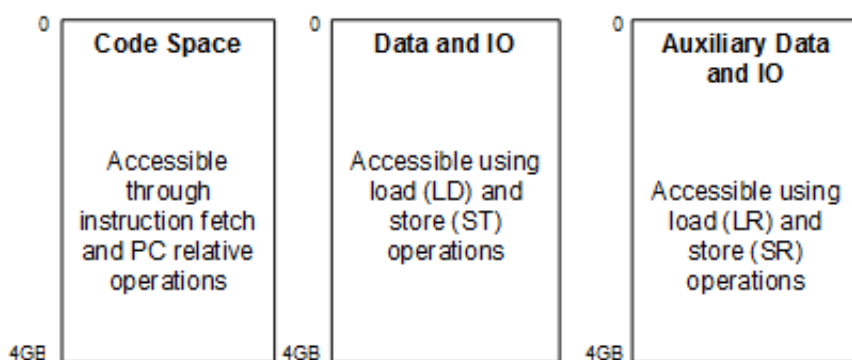
This chapter describes the data organization and addressing used by the ARCV2-based processor.

4.1 The ARCV2 Address Space

An ARCV2 processor supports the following three distinct address spaces:

- A half-word addressable Instruction Address Space
- A byte-addressable Data Address Space.
- A 32-bit Auxiliary address space. This address space provides an additional 4G word locations that are typically used to access the Control and Status registers, I/O devices, Build Configuration registers, and Application-specific Customer Extension registers. This space is word-addressable using LR, SR and AEX instructions. See [Figure 4-1](#) and [4.2](#) for more information.

Figure 4-1 Address Space Model



All ARCV2-based processors have physically independent Instruction and Data paths that allow for von Neumann or Harvard configurations. However, the default memory configuration for the processor unifies the Data and Instruction memory spaces. A load or store to the memory address location *nn* in the data memory accesses location *nn* in the instruction memory.

4.2 Data Formats

All ARCV2-based processors support little-endian byte ordering by default. Some ARCV2 processors can be configured to be big-endian.

The processor can operate on data of various sizes. The memory operations (load and store type operations) can have data of 32 bits (Word), 16 bits (Half-word), or 8 bits (Byte). When LL64_OPTION is supported and enabled, 64-bit data can also be accessed as a pair of 32-bit words.

Byte operations use the least significant 8 bits of the processor register from which data is stored, or to which data is loaded. Byte loads may extend the sign of the byte across the rest of the word depending on the load instruction. The same applies to the half-word operations with the half-word occupying the least-significant 16 bits of the register involved in a load or store of half-word objects.

Double-word (64-bit) operations use a pair of registers as the source of store data and the destination of load data. Such register pairs are defined by an even-numbered register R_n and implicitly use R_{n+1} . Which register within the register pair holds the most significant word depends on the endian configuration of the processor. An illegal instruction exception is raised if an instruction attempts to access a register pair operand using an odd-numbered register. For more information, see [Figure 4-5](#) and [Figure 4-6](#).

4.2.1 Vector Operands

Vector instructions operate on multiple data elements stored within registers. These instructions use the following nomenclature when referring to data elements within a 32-bit register or a 64-bit register pair.

4.2.1.1 64-Bit Operand Data Elements

64-bit source operands are possible only when $MPY\_OPTION > 6$, $LL64\_OPTION == 1$, or $FPU\_HAS\_DP == 1$. A 64-bit operand has a most-significant-word, w1 or $OPD.w1$, and a least-significant-word, w0 or $OPD.w0$.

Figure 4-2 64-bit Element As a 64-bit Operand

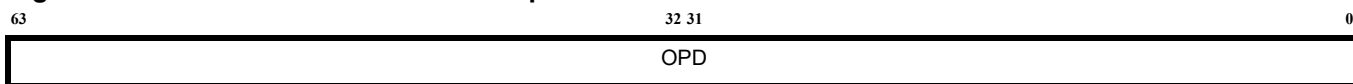


Figure 4-3 32-bit Elements in a 64-bit Operand

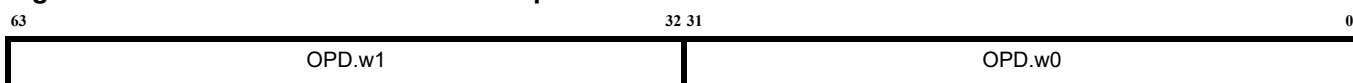
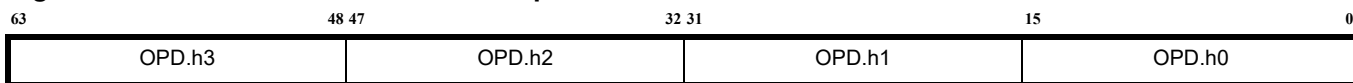


Figure 4-4 16-bit Elements in a 64-bit Operand



Double-Word Operations

Double-word (64-bit) operations use a pair of registers for source and/or destination data. Such register pairs are defined by an even-numbered register R_n and implicitly use R_{n+1} .

Figure 4-5 Register Pair Holding Operands in Little-endian Mode

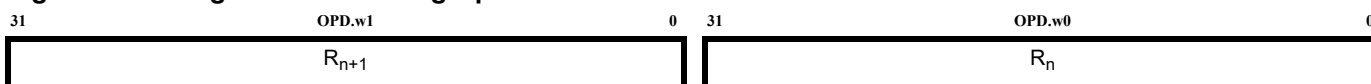
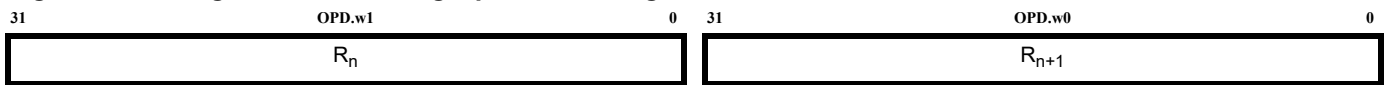


Figure 4-6 Register Pair Holding Operands in Big-endian Mode

ARCV2 processors maintain little-endian byte ordering within each register. However, in ARC32 mode, when a register pair is used to construct a 64-bit operand, the register numbering depends on the endian mode configured for the processor.

4.2.1.2 LDD and STD Operands

The Load Double (LDD) and Store Double (STD) instructions are provided when LL64_OPTION is enabled. These instructions load (or store) a pair of 32-bit registers from (or to) memory. Effectively these instructions move a vector of two 32-bit objects between a pair of registers and memory. The ARC 32-bit ABI stores 64-bit objects in memory as a vector of two 32-bit values, and therefore LDD and STD can be used to access 64-bit objects that are laid out in memory according to the ARC 32-bit ABI.

When LDD and STD instructions are executed, the low memory address always goes with the even register R_n and the high memory address (low+4) goes with the odd register R_{n+1} .

Consider the following double load instruction:

```
LDD Rn, [A]
```

Little-endian mode (LE)

$$R_{n+1} \leftarrow \{ A+7, A+6, A+5, A+4 \}$$

$$R_n \leftarrow \{ A+3, A+2, A+1, A \}$$

Big-endian mode (BE)

$$R_{n+1} \leftarrow \{ A+4, A+5, A+6, A+7 \}$$

$$R_n \leftarrow \{ A, A+1, A+2, A+3 \}$$

The values in R_n and R_{n+1} are the same as they would be if the same 64-bit value had been passed as a function argument using two consecutive registers, R_n and R_{n+1} . The 32-bit word at address A is passed through R_n , and the 32-bit word at address $A+4$ is passed through R_{n+1} , regardless of the byte ordering of the machine.

If a 64-bit object has been loaded from memory, the most-significant word is in R_n for a big-endian system, or R_{n+1} for a little-endian system. Likewise, the least-significant word is in R_{n+1} for a big-endian system, or R_n for a little-endian system. Therefore, instructions that operate on 64-bit register pairs must respect this endian-specific register-pair ordering.

4.2.2 32-Bit Operand Data Elements

Figure 4-7 32-Bit Elements in a 32-bit Operand

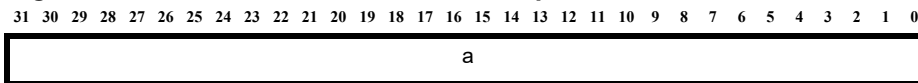
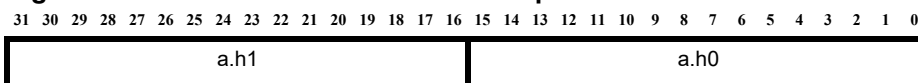


Figure 4-8 16-Bit Elements in a 32-bit Operand



4.2.3 Expansion of Literals

The `u6` and `s12` instruction literals are referred to as short-immediate (or `shimm`) operands. When short-immediate (`shimm`) or long-immediate (`limm`) operands are used in a 64-bit context (except `STD`) or in vector arithmetic instructions, the following diagrams illustrate the expansion of the literals to the operand size appropriate to the instruction.

Figure 4-9 and Figure 4-10 illustrate expansion of a `shimm` to a word (32 bits).

Figure 4-9 Expansion of a `u6` Literal to a 32-Bit Word (`w-shimm`)

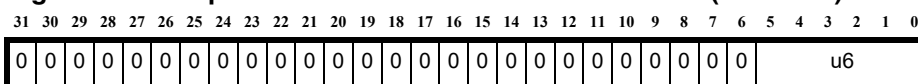


Figure 4-10 Expansion of an `s12` Literal to a 32-Bit Word (`w-shimm`)

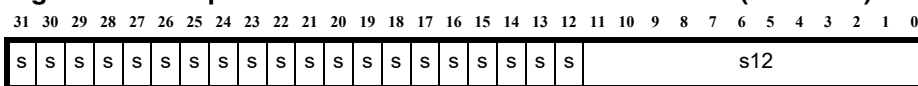


Figure 4-11 and Figure 4-12 illustrate expansion of a `shimm` to a half-word (16-bits).

Figure 4-11 Expansion of a `u6` Literal to Half-Word (16-bit, `hw-shimm`)

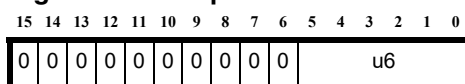
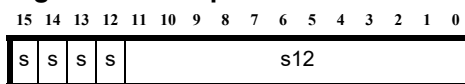


Figure 4-12 Expansion of an `s12` Literal to Half-Word (16-bit, `hw-shimm`)



4.2.4 Expansion of Literals in Vector Instructions

When a vector instruction expects 16-bit elements as a source operand, the `shimm` is replicated to the required operand size as shown in the following table:

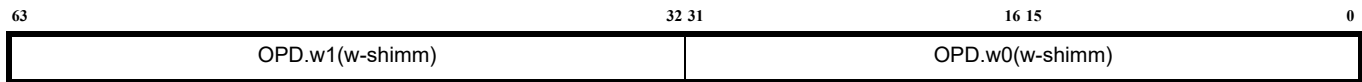
Figure 4-13 A Half-Word (16-bit) Short Limm Value Replicated to form a 64-Bit Operand

63	48 47	32 31	16 15	0
OPD.h3 (hw-shimm)		OPD.h2 (hw-shimm)		OPD.h1 (hw-shimm)
				OPD.h0 (hw-shimm)

Where a `hw-shimm` represents `u6` or `s12` extended to half-word (16-bits). For an illustration of `hw-shimm`, see Figure 4-11 and Figure 4-12.

When a vector instruction expects 32-bit elements as a source operand, the w-shimm is replicated to the required operand size as shown in the following table:

Figure 4-14 A word (32-bit) Short-limm or Limm value Replicated to form a 64-Bit Operand



Where a w-shimm represents u6 or s12 expanded to a word (32-bits), or a limm. For an illustration of w-shimm, see [Figure 4-9](#) and [Figure 4-10](#).

[Figure 4-15](#) and [Figure 4-16](#) illustrate the expansion of two hw-shimm(16 bits) to a word (32 bits).

Figure 4-15 Two 16-Bit Vectors with Expanded u6 Operands

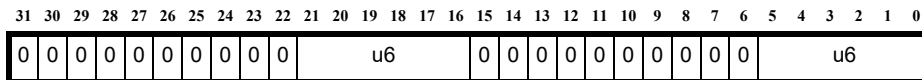
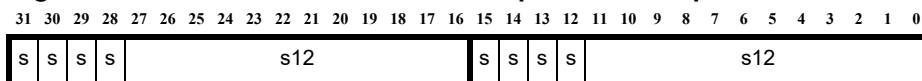


Figure 4-16 Two 16-Bit Vectors with Expanded s12 Operands



[Figure 4-17](#) and [Figure 4-18](#) illustrate the expansion of four 8-bits vectors (expanded u6 or s12 literals) to a word (32 bits).

Figure 4-17 Four 8-Bit Vectors with Expanded u6 Operands

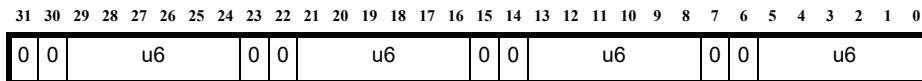


Figure 4-18 Four 16-Bit Vectors with Expanded s12 Operands

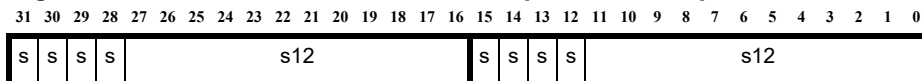
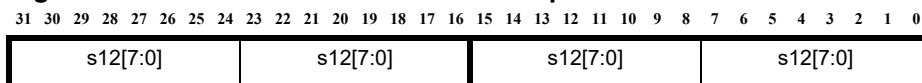


Figure 4-19 Four 8-Bit Vectors with s12 Operands

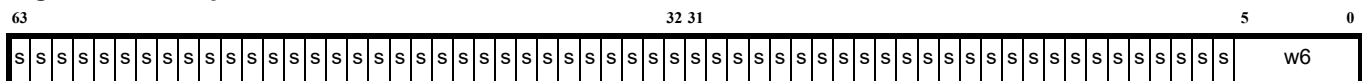


4.2.5 Expansion of Literals in Store Double (STD)

The w6 store-data operand for an STD instruction is always sign-extended to 64 bits before being stored into memory. If the store-data operand for an STD is a LIMM, OPD.w1 is the sign-extension of OPD.w0.

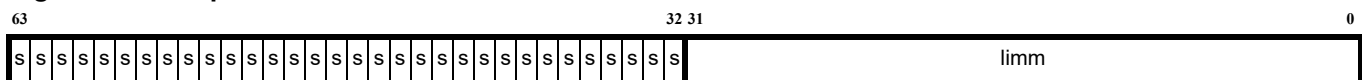
[Figure 4-20](#) illustrates the expansion of a w6 in STD.

Figure 4-20 Expansion of a w6 in STD



[Figure 4-21](#) illustrates the expansion of a limm in STD

Figure 4-21 Expansion of a limm in STD



4.3 Floating-Point Data Formats

The floating-point unit supports the following data formats:

- “Single-Precision Floating-Point Data Formats” on page 120
- “Double-Precision Floating-Point Data Formats” on page 120
- “Double Precision Floating-Point Data Formats” on page 121

4.3.1 Single-Precision Floating-Point Data Formats

The format of the source operands and result are as specified by the IEEE-754 standard.

A single-precision floating-point number is represented in a core register as shown in [Figure 4-22](#).

Figure 4-22 Single-Precision Floating-Point Data

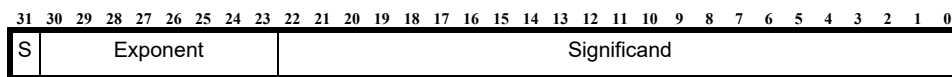


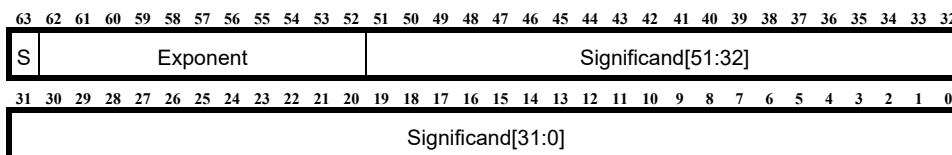
Table 4-1 Single-Precision Floating-Point Data Field Descriptions

Field	Description
S	Sign part of the floating-point number 0—Positive number 1—Negative number
Exponent	Exponent part of the floating-point number
Significand	Significand part of the floating-point number

4.3.2 Double-Precision Floating-Point Data Formats

When hardware support for double-precision floating-point operations is configured (FPU_DP_OPTION == 1), a double-precision floating-point number is represented in core registers as shown in [Figure 4-23](#).

Figure 4-23 Double-Precision Floating-Point Data



The bit fields are described in the following table.

Table 4-2 Double-Precision Floating-Point Data Field Descriptions

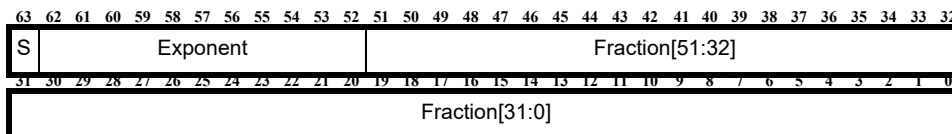
Field	Description
S	Sign part of the floating-point number 0: Positive Number 1: Negative Number
Exponent	Exponent part of the floating-point number
Significand	Significand part of the floating-point number

4.4 Double Precision Floating-Point Data Formats

When the FPU_DP_ASSIST option is configured, double-precision floating-point instructions use the following auxiliary registers to represent 64-bit double-precision data:

- [AUX_DFP1L](#)
- [AUX_DFP1H](#)
- [AUX_DFP2L](#)
- [AUX_DFP2H](#)

The 64-bit double precision floating-point data registers, D1 (AUX_DFP1L and AUX_DFP1H) and D2 (AUX_DFP2L and AUX_DFP2H), use the IEEE-754 standard format as shown in [Figure 4-24](#).

Figure 4-24 Double Precision Floating-Point Data**Table 4-3 Double Precision Floating-Point Data Field Descriptions**

Field	Description
S	Sign part of the floating-point number 0—Positive number 1—Negative number
Exponent	Exponent part of the floating-point number
Fraction	Fractional part of the floating-point number

4.4.1 NaN Formats

The floating-point unit supports IEEE 754 defined NaNs (Not-a-number): signaling (SNaN) and quiet (QNaN). Signaling NaNs can be used by software, for example for uninitialized variables. Operations that signal an invalid operation but still need to return a floating-point result return a QNaN.

The floating-point unit handles the NaNs as follows:

- If a NaN (quiet or signaling) is supplied as an input to a floating-point operation, the result of that operation is the input NaN. However, if the input is an SNaN, the output is a QNaN. The SNaN is converted to a QNaN by setting the most significant bit of the significand to 1.
- For dual operand floating-point instructions, if both source values are NaNs, the result is selected from the inputs with the following priority order:
 - Source operand 0 if it is an SNaN
 - Source operand 1 if it is an SNaN
 - Source operand 0 if it is a QNaN
 - Source operand 1 if it is a QNaN

For fusedMultiplyAdd and fusedMultiplySubtract floating-point instructions, the following rules apply:



Note

For the fusedMultiplyAdd and fusedMultiplySubtract floating-point instructions, source operand 2 is the accumulator.

- Source operand 2 if it is an SNaN
- Source operand 0 if it is an SNaN
- Source operand 1 if it is an SNaN
- Source operand 2 if it is a QNaN



Note

If source operand 0 and source operand 1 are in the following combinations:

$0 * \text{infinity}$

or

$\text{infinity} * 0$

the default NaN is returned and the instruction raises an invalid flag.

- Source operand 0 if it is a QNaN
- Source operand 1 if it is a QNaN.

4.5 Data Layout in Memory

Baseline ARCV2 architectures access data memory using byte addresses and require that all memory addresses are aligned as follows:

- 64-bit Double-words are aligned to 32-bit word boundaries (requires LL64_OPTION)
- 32-bit Words are aligned to 32-bit word boundaries
- 16-bit Half-words are aligned to 16-bit half-word boundaries
- Bytes have no specific alignment

In a baseline ARCv2 architecture, all misaligned data accesses raise a data misalignment exception. Certain high performance versions of the ARCv2 architecture can access memory using non-aligned accesses. Software can detect whether non-aligned memory accesses are supported by examining the `NON_ALIGNED` field of the [Instruction Set Configuration Register, ISA_CONFIG](#) build configuration register. When this field is set to 1, the implementation supports non-aligned memory accesses.

When non-aligned memory references are supported, an additional Alignment Disable Control bit (AD) is present in bit position 19 of the STATUS32 ([Status Register, STATUS32](#)) register. This bit also appears in the interrupt and exception copies of STATUS32 (STATUS32_P0 and ERSTATUS). When STATUS32.AD is set to 1, the processor's non-aligned referencing capabilities are enabled. When it is set to 0, the processor enforces natural alignment just as it would do in a baseline processor. The reset value of STATUS32.AD is 0, and therefore software must explicitly enable non-aligned access capabilities by setting this bit to 1, when required. In a baseline processor, the AD bit is not present in STATUS32, and is therefore read as zero and ignored on write.

4.5.1 64-Bit Data

[Figure 4-25](#) shows the -little-endian representation in byte-wide memory. If the ARCv2-based processor supports -big-endian addressing, the data is stored in memory as shown in [Figure 4-26](#)

Figure 4-25 64-bit Register Data in Byte-Wide Memory, Little-Endian

Address	7	6	5	4	3	2	1	0
N	Byte 0							
N+1	Byte 1							
N+2	Byte 2							
N+3	Byte 3							
N+4	Byte 4							
N+5	Byte 5							
N+6	Byte 6							
N+7	Byte 7							

Figure 4-26 64-bit Register Data in Byte-Wide Memory, Big-Endian

Address	7	6	5	4	3	2	1	0
N	Byte 7							
N+1	Byte 6							
N+2	Byte 5							
N+3	Byte 4							
N+4	Byte 3							
N+5	Byte 2							
N+6	Byte 1							
N+7	Byte 0							

For information about the 64-bit data organization in a register pair, see [Figure 4-5](#) and [Figure 4-6](#).

4.5.2 32-Bit Data

All load or store, arithmetic, and logical operations support 32-bit data. [Figure 4-27](#) on page 124 show the data representation in a general purpose register. [Figure 4-28](#) on page 124 shows the little-endian representation in byte-wide memory. [Figure 4-29](#) on page 124 shows the big-endian representation.

Figure 4-27 Register containing 32-bit Data

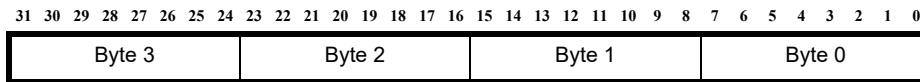


Figure 4-28 32-bit Register Data in Byte-Wide Memory, Little-Endian

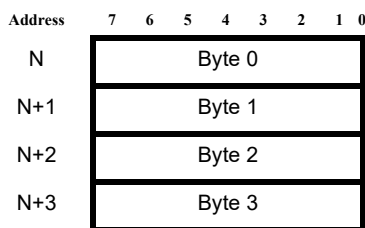
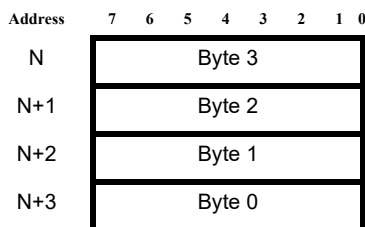


Figure 4-29 32-bit Register Data in Byte-Wide Memory, Big-Endian



4.5.3 16-Bit Data

Load, store, and some multiplication instructions support 16-bit data. 16-bit half-word data can be promoted explicitly to an equivalent 32-bit word value by using unsigned extend (EXTH (see [EXTH](#))) or signed extend (SEXH (see [SEXH SEXW](#))) instructions. [Figure 4-30](#) on page 125 shows the 16-bit data representation in a general purpose register.

For the programmer's model, the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a *data invariance* principle. [Figure 4-31](#) on page 125 shows the little-endian representation of 16-bit data in byte-wide memory. If the

ARCV2-based processor supports big-endian addressing, [Figure 4-32](#) on page 125 shows how the 16-bit data is stored in memory.

Figure 4-30 Register Containing 16-Bit Data

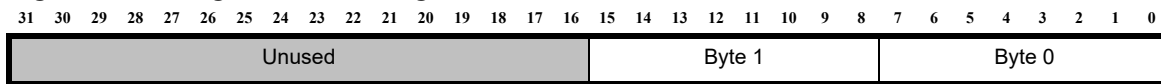


Figure 4-31 16-bit Register Data in Byte-Wide Memory, Little-Endian

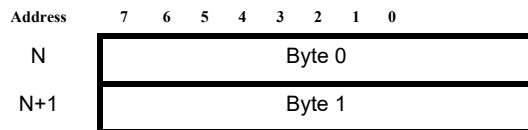


Figure 4-32 16-bit Register Data in Byte-Wide Memory, Big-Endian



4.5.4 8-Bit Data

The load and store operations support 8-bit data which can be promoted to a 32-bit value by using unsigned extend (EXTB) or signed extend (SEXB) instructions. [Figure 4-33](#) on page 125 shows the 8-bit data representation in a general purpose register.

For the programmer's model, the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a *data invariance* principle. [Figure 4-34](#) on page 125 shows the representation of 8-bit data in byte-wide memory.

Regardless of the endianness of the ARCV2-based system, the byte-aligned address, N, of the byte is explicitly given and the byte is stored or read from that explicit address.

Figure 4-33 Register Containing 8-Bit Data

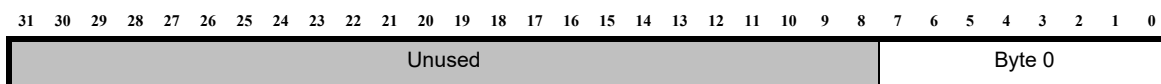
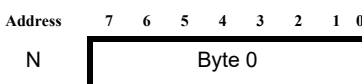
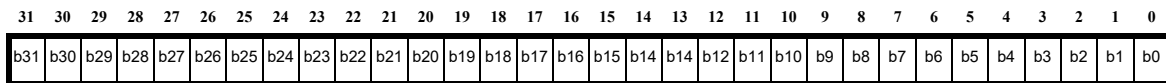


Figure 4-34 8-bit Register Data in Byte-Wide Memory



4.5.5 1-Bit Data

The ARCV2 instruction-set architecture supports single-bit operations on data stored in the core registers. A bit manipulation instruction includes an immediate value specifying the bit to operate on. Bit manipulation instructions can operate on 8-bit, 16-bit, or 32-bit data located within core registers because each bit is individually addressable.

Figure 4-35 Register Containing 1-Bit Data

4.6 Instruction Layout in Memory

The ARCV2 instruction set supports freely intermixed 16-bit and 32-bit instructions, each of which may have an additional 32-bit immediate literal value (referred to as long-immediate data throughout this document)

All programs are represented as a sequence of 16-bit half-words. A 32-bit instruction has two such half-words. A 32-bit long-immediate data value is similarly represented as a sequence of two half-word values. A 16-bit instruction is encoded as a single half-word.

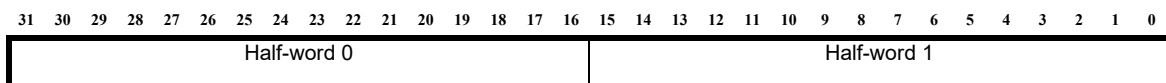
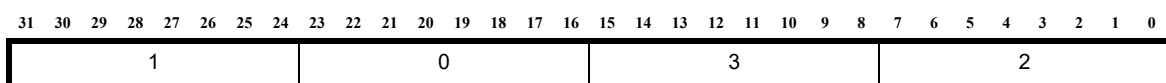
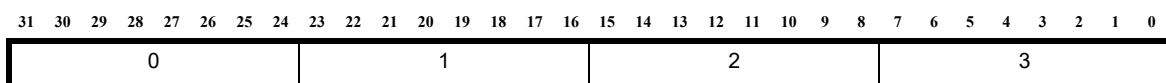
Figure 4-36 Half-word Numbering of a 32-Bit Instruction or Long-immediate Data Item

Figure 4-36 shows the number of two half-words in a 32-bit instruction or long-immediate data item.

If an instruction has a long-immediate data item, it follows in the two consecutive half-words after the instruction.

The location of each byte of an instruction or long-immediate value depends on the memory byte ordering configured for the processor. An ARCV2-based processor can be configured to support either big-endian or little-endian byte ordering. This configuration is always static, and cannot be altered at runtime.

Figure 4-37 illustrates the little-endian memory address offset of each byte, within a 32-bit instruction or long-immediate value, relative to the start address of that instruction or long-immediate value. Figure 4-38 shows the corresponding big-endian addresses.

Figure 4-37 Little-endian Offset of each Byte in a 32-Bit Instruction or Immediate**Figure 4-38 Big-endian Offset of each Byte in a 32-Bit Instruction or Immediate**

Similarly, Figure 4-39 illustrates the little-endian memory address offset of each byte within a 16-bit instruction, relative to the start address of that instruction. Figure 4-40 illustrates the big-endian memory offset of each byte in a 16-bit instruction.

4.6.1 Addressing Modes

Figure 4-39 Little-endian Memory Offset of each Byte in a 16-Bit Instruction

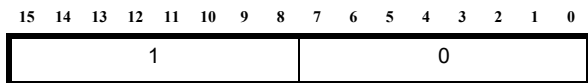
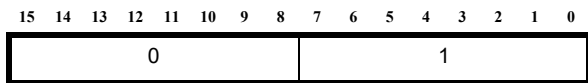


Figure 4-40 Big-endian Memory Offset of each Byte in a 16-bit Instruction



The following six basic addressing modes are supported by the architecture:

Register Direct	Operations are performed on values stored in the registers
Register Indirect	Operations are performed on locations specified by the contents of registers.
Register Indirect with offset	Operations are performed on locations specified by the contents of a register and an offset value (in another register, or as immediate data)
Immediate	Operations are performed by using the constant data stored within the opcode.
PC relative	Operations are performed relative to the current value of the Program Counter (usually branch or PC relative loads).
Absolute	Operations are performed on the data at a location in memory specified by a constant value in the opcode.

The following sections describe the instruction formats for each addressing mode. The descriptions use one of these formats. An instruction is described by the operation (op), including optional flags, the operand list.

Table 4-4 Instruction Variants

Operation	
<.f>	write-back to status register flags
<.cc>	condition code field (for example, conditional branch)
<.d>	delay slot follows instruction (used for branch and jump)
<.zz>	size definition (byte, half-word, or word)
<.x>	perform sign extension
<.di>	data cache bypass (load and store operations)
<.aa>	address write-back
<.T>	invert the default static branch prediction (see “Static Branch Prediction Mode” on page 362)

Table 4-5 Operand Description

Operand	
a, b, c	General Purpose registers, reduced range for 16-bit instructions. When the encoded value of the source operands is 62, the instruction indicates long-immediate data.
h, g	General Purpose register, capable of addressing registers r0-r28 and r31 in some of the 16-bit encoded instructions. When these operands encoded value is 30, the instruction indicates long-immediate data rather than r30. And similarly, operand encoded value of 29 is reserved.
u<X>	Unsigned immediate values of size <X>-bits
s<X>	Signed immediate values of size <X> bits
limm	Long immediate value of size 32-bits (stored as a second opcode)

4.6.2 Null Instruction Format

The ARCV2 ISA supports a special type of instruction format, where the destination of the operation is defined as null (0). When this instruction format is used, the result of the operation is discarded, but the status flags may be set depending on <.f> or the instruction. This design allows any instruction to act in a manner similar to compare.

Example 4-1 Null Instruction Format

```

ADD.F    r1, r2, r3      ;Normal syntax
                        ;the result of r2+r3
                        ;is written to r1 and
                        ;the flags are updated
ADD.F    0, r2, r3       ;Null syntax
                        ;the result of r2+r3 is
                        ;used to update the
                        ;flags, but is not saved.
MOV      0, 0            ;Null syntax
                        ;recommended NOP equivalent

```

Because all 32-bit instruction formats support this mode, a 32-bit NOP is not explicitly defined. However, the recommended NOP_L equivalent is MOV 0, 0. The 16-bit instruction set provides a no-operation instruction, NOP_S.

4.6.3 Conditional Execution

A number of the 32-bit instructions in the ARCV2 ISA support conditional execution. A 5-bit condition code field allows up to 32 independent conditions to be tested before execution of the instruction. By default, 16 conditions are defined with the remainder available for customer definition, as required.

4.6.4 Conditional Branch Instruction

The 32-bit and 16-bit instructions support conditional branch (Bcc) operations. The 32-bit instructions also include conditional jump and jump and link (Jcc and JLcc, respectively) whereas the 16-bit instruction set provides only unconditional jumps.

4.6.5 Compare and Branch Instruction

The ARCV2 ISA includes two forms of instruction which integrate compare/test and branch.

The Compare and Branch Conditionally (BRcc) command is the juxtaposition of compare (CMP) and conditional branch (Bcc) instructions. These instructions are available in both 32-bit and limited 16-bit versions.

The 'Branch if bit set/clear' (BBIT0, BBIT1) instructions provide the operation of the bit test (BTST) and 'Branch if equal/not equal' (Bcc) instructions. These instructions are only available as 32-bit instructions.

Core Architectural State Registers

Figure 5-1 shows a summary of the core register set.

	31																																1	0
r0																																		
r1																																		
-	32-bit Basecase Core Registers																																	
r24																																		
r25																																		
r26	Global Pointer (GP)																																	
r27	Frame Pointer (FP)																																	
r28	Stack Pointer (SP)																																	
r29	Interrupt Link Register (ILINK)																																	
r30	General Purpose Register																																	
r31	Branch Link Register (BLINK)																																	
r32																																		
r57	32-bit Extension Core Registers																																	
r57																																		
r58	Accumulator Low ACCL (little endian), ACCH (big-endian)																																	
r59	Accumulator High- ACCH (little endian), ACCL (big endian)																																	
r60	LP_COUNT[LPC_SIZE-1:0]																																	
r61	RESERVED																																	
r62	Long Immediate Data Indicator																																	

Figure 5-1 Core Register Map Summary (Continued)

r63	Program Counter [PC_SIZE-1:2], Read-Only (PCL)	0	0

The default implementation of the core provides 32 general purpose 32-bit core registers. If required, up to 26 extension core registers can be defined, using APEX, when the core is configured. Alternatively, the core can be de-configured to provide only 16 general purpose 32-bit core registers. To reduce the physical size and power consumption of the core. When executing instructions encoded in 32-bit formats, the full range of core registers is available. [Table 5-1](#) on page 132 lists summarizes the accessibility of core registers from 16-bit and 32-bit instruction formats.

Table 5-1 Core Register Set

Register	32-bit Instruction Function and Default Usage	16-bit Instruction Access to Register
r0	General purpose	Default access
r1	General purpose	Default access
r2	General purpose	Default access
r3	General purpose	Default access
r4 - r11	General purpose	MOV_S, CMP_S, ADD_S
r12	General purpose	Default access
r13	General purpose	Default access
r14	General purpose	Default access
r15	General purpose	Default access
r16 - r25	General purpose	MOV_S, CMP_S, ADD_S
r26 (GP) (see Pointer Registers, GP (r26), FP (r27), SP (r28))	Global pointer	LD_S, MOV_S, CMP_S, ADD_S
r27 (FP) (see Pointer Registers, GP (r26), FP (r27), SP (r28))	Frame pointer (default)	MOV_S, CMP_S, ADD_S
r28 (SP) (see Pointer Registers, GP (r26), FP (r27), SP (r28))	Stack pointer	PUSH_S, POP_S, SUB_S, LD_S, ST_S, MOV_S, CMP_S, ADD_S
r29 (ILINK) (see Link Registers, ILINK (r29), BLINK (r31))	Interrupt link	
r30	General purpose	Default access. Long immediate indicator for instructions with 'h' operand

Table 5-1 Core Register Set (Continued)

Register	32-bit Instruction Function and Default Usage	16-bit Instruction Access to Register
r31 (BLINK) (see Link Registers, ILINK (r29), BLINK (r31))	Branch link register	JL_S, BL_S, J_S, PUSH_S, POP_S, MOV_S, CMP_S, ADD_S
r32 - r57	Extension core registers	
r58	Accumulator	
r59	Accumulator	
r60 (LP_COUNT) (see Loop Count Register, LP_COUNT (r60))	Loop counter	
r61	Reserved	Reserved
R62	Long immediate	MOV_S, CMP_S, ADD_S
R63 (PCL) (see Word-aligned Program Counter, PCL (r63))	Program counter (32-bit aligned)	

5.1.1 Hardware Initialization of Register File

When you set the configuration option `rf_rst` to `true`, the global `rst_a` signal is connected to the core register file and resets the data in the register file to zero.

5.1.2 Core Register Mapping Used in 16-bit Instructions

The 16-bit instructions use only 3 bits for register encoding. However, the 16-bit move (MOV_S (see [MOV](#))), compare (CMP_S (see [CMP](#))), and add (ADD_S (see [ADD](#))) instructions can access a wider set of core registers. This design facilitates the copying and manipulation of data stored in registers that are not accessible to other 16-bit instructions.

The most-frequently used registers, according to the ARCV2 application binary interface (ABI), are r0-r3 (ABI call argument registers), r12 (temporary register), and r16-r25 (ABI call saved registers). [Table 5-2](#) on page 133 lists the special register encoding and [Table 5-3](#) on page 134 lists the ABI usage support.

Table 5-2 16-bit Instruction Register Encoding

16-bit Instruction Register Encoding	32-bit Instruction Register
0	r0
1	r1
2	r2

Table 5-2 16-bit Instruction Register Encoding (Continued)

16-bit Instruction Register Encoding	32-bit Instruction Register
3	r3
4	r12
5	r13
6	r14
7	r15

5.1.3 Reduced Configuration of Core Registers

The ARCV2 ISA supports a reduced set of 16 core registers. To support the ARCV2-based ABI, the set of reduced registers is indicated in [Table 5-3](#) on page 134. The RF_BUILD register is used to determine the configuration of core registers (see [Core Register File Configuration Register, RF_BUILD](#)).

Table 5-3 Current ABI Register Usage

Register	Use	16-bit Instruction Access	Reduced Configuration
r0-r3	argument regs	•	•
r4-r7	argument regs		
r8-r9	temp regs		
r10-r11	temp regs		•
r12-r15	temp regs	•	•
r16-r25	saved regs		
r26	GP (global pointer) (see Pointer Registers, GP (r26), FP (r27), SP (r28))		•
r27	FP (frame pointer) (see Pointer Registers, GP (r26), FP (r27), SP (r28))		•
r28	SP (stack pointer) (see Pointer Registers, GP (r26), FP (r27), SP (r28))		•
r29	ILINK (see Link Registers, ILINK (r29), BLINK (r31))		•
r30	General purpose register Note: This register is designated neither as a saved nor a temp register.		•
r31	BLINK (see Link Registers, ILINK (r29), BLINK (r31))		•

5.1.4 Multiple Register Banks

The ARC EM can be hardware configured with more than one register file banks using the `rgf_num_banks` configuration option. See the *DesignWare ARC EM Databook* for more information. Additionally, the number of registers in each register bank is configurable



Note

When `-sec_modes_option==true`, multiple register banks are not supported.

This option defines the number of register file banks in the CPU. When using more than one register file bank, one should configure the number of registers and also specify the registers that must be replicated in each register bank. The currently-selected bank number is defined by the three-bitfield `STATUS32.RB[2:0]`.



Note

The following table shows all of the possible register file configurations for ARC EM. If you are working with an already generated ARC EM core, the Core Register File Configuration Register, `RF_BUILD`, contains information regarding the number of register banks and the number of registers in each bank. See [RF_BUILD](#) register for more information.

Table 5-4 Multiple Bank Configuration

RGF_NUM_BANKS	RGF_NUM_REGS	RGF_BANKED_REGS	Bank0	Bank1
1	16		R0-R3, R10-R15, R26-R31	
	32		R0-R31	
2	16	4	R0-R3, R10-R15, R26-R31	R0-R3
		8	R0-R3, R10-R15, R26-R31	R0-R3, R12-R15
		16	R0-R3, R10-R15, R26-R31	R0-R3, R10-R15, R26-R28, R30-R31
	32	4	R0-R31	R0-R3
		8	R0-R31	R0-R3, R12-R15
		16	R0-R31	R0-R3, R10-R15, R26-R28, R30-R31
		32	R0-R31	R0-R28, R30-31

When multiple register banks are present in the core ($\text{RGF\_NUM\_BANKS} > 1$), the number of general purpose registers present in the extra banks is also configurable. The RGF_BANKED_REGS option defines the number of baseline core registers that are contained in banks other than bank 0 (replicated banks).

- In configurations with 32 base GP registers (RF32 configurations), 4, 8, 16, or 32 GP registers can be populated in the extra register banks, using the RGF_BANKED_REGS configuration option.
- In configurations with 16 base GP registers (RF16 configurations), 4, 8, or 16 GP registers can be populated in the 'extra' register banks, using the RGF_BANKED_REGS configuration option.

5.1.4.1 Accessing Register Banks



Before accessing register banks, ensure that you have configured your processor to include multiple register banks. For more information, see section [Multiple Register Banks](#).

The three-bitfield, $\text{STATUS32.RB}[2:0]$, indicates the register bank that is currently selected. Use the following procedure to select a register bank and write to it:

In kernel mode, use the KFLAG instruction to write to the $\text{STATUS32.RB}[2:0]$ to select a register bank. Any access to core registers from thereon are made to the register in the selected register bank.

When RGF_NUM_BANKS is set to 1, STATUS32.RB field is read as zero and ignored on writes.

Second Bank SP and GP Initialization

The Metaware and GCC initialization code do not initialize the Stack pointer of the 'shadow' register files. You (user code (and linker files)) must do the following:

- Define the additional stack regions and sda_base to be used for the shadow register files.
- Initialize the shadow register's stack pointer and sda_base registers.

Example:

```
//declaration of linker symbols
extern unsigned char _estack2[]; //end of stack2;
extern unsigned char SDA_BASE[]; //Small-data base register

//Setup registers in second bank (bank 1):
//1. GP register for SDA Base
//2. SP register for stack2 end (desired stack for bank 1)
// This MUST be done before any shadow register file C code is executed
void setupBank1(void)
{
    unsigned int stat32 = _lr(STATUS32);
    //switch to register bank 1 (the duplicate register bank):
    uint32_t stat32 &= ~(1 << 16);
    stat32 |= 1 << 16;
    _kflag(stat32);
    _core_write((unsigned int)_estack2, REG_SP);
    _core_write((unsigned int)_SDA_BASE_, REG_GP);
}
```



```

//switch to the 'primary' register bank (register bank 0)
stat32 = _lr(STATUS32);
stat32 &= ~(1 << 16);
_kflag(stat32);
_nop();
}

```

5.1.4.2 Debugger Access to Alternate Register Banks

The debug host can enable an independent selection of which register bank to access when accessing core registers. Set the DEBUGI.RBE bit and write to DEBUGI.RB field to select a register bank in debug mode.



Note

If DEBUGI.RBE is set, the DEBUGI.RB field preempts the STATUS32.RB field for all instructions originating from the debug interface. When DEBUGI.RBE is not set, the existing value from the STATUS32.RB field is used.

5.1.5 Illegal Core Register Usage

An [Illegal Instruction](#) exception is raised in the following instances:

- References to an unimplemented core register
- Writes to a read-only register
- Reads from a write-only core register

See also “[Illegal Extension Core Register Usage](#)” on page 140.

5.1.6 Pointer Registers, GP (r26), FP (r27), SP (r28)

The ARCV2 application binary interface (ABI) defines 3 pointer registers: Global Pointer (GP), Frame Pointer (FP), and Stack Pointer (SP), which use registers r26, r27, and r28, respectively. The global pointer (GP) is used to point to small sets of shared data throughout execution of a program. The stack pointer (SP) register points to the lowest-used address of a downward-growing stack. The frame pointer (FP) register points to an ABI-defined base address for the current stack frame. The ABI usage of core registers is summarized in [Table 5-3](#) on page 134.

5.1.7 Link Registers, ILINK (r29), BLINK (r31)

The link registers (ILINK and BLINK) are used to provide links back to the position where an interrupt, branch-and-link, or jump-and-link occurred.

The ILINK register may be modified on entry to an interrupt of any level, and may also be modified on exit from an interrupt at any level. Therefore, ILINK cannot be relied upon to retain its value unless executing within a level 0 interrupt handler, or within an exception handler (when interrupts are also disabled), or if all interrupts are disabled.

The ILINK register is not accessible in user mode. Illegal accesses from user mode to ILINK raise a Privilege Violation exception (see [Privilege Violation, Kernel/User Only Access](#)).

The cause code for this violation is indicated in the Exception Cause register (see [Exception Cause Register, ECR](#)). When there are multiple register banks in the CPU, and the CPU is configured to replicate 16 or 32 registers in the duplicate register banks, the ILINK register is not replicated in the additional register banks.

r31 is the BLINK register. Returning from a branch-and-link (BLcc) or jump-and link (JLcc) is accomplished by jumping to the contents of the BLINK register, using the Jcc [BLINK] instruction (see [Jcc](#)).

If the delay-slot instruction for any of the branch or jump instructions that set BLINK (BL.D, BLcc.D, JL.D, or JLcc.D) uses a long-immediate operand, the link value assigned to BLINK by the preceding branch or jump-and-link instruction becomes unpredictable. It is illegal for an instruction appearing in the delay-slot of a branch or jump instruction to have a long-immediate data operand (LIMM) and the processor raises an Illegal Instruction Sequence exception. The processor ignores any such LIMM operand specified by a delay-slot instruction, and consequently determines the size of the delay-slot instruction based solely on its major opcode. If the preceding branch or jump instruction sets the BLINK register (such as JL.D or BL.D), then the BLINK value defined by that instruction is the address of its delay-slot instruction plus the size of its delay-slot instruction, but always excluding any illegal LIMM operand.

5.1.8 Long Immediate Data Indicator Register, (r30)

r30 is a general purpose register that can be used as a long immediate data indicator for h operand instructions.

5.1.9 Loop Count Register, LP_COUNT (r60)

The loop count register (LP_COUNT) is used for zero delay loops. Do not use LP_COUNT as a general purpose register in your own assembly code because LP_COUNT maybe decremented if the program counter equals the loop end address. However, the MetaWare compiler can safely use this register as a general purpose register. For more information about the zero delay loop mechanism, see [LP<cc>](#) instruction details on [page 622](#).

The LP_COUNT register must not be used as the destination of multi-cycle instruction, including multi-cycle extension instructions. An intermediate register must be used to convey the result to LP_COUNT. A multi-cycle instruction writing to the LP_COUNT register raises an [Illegal Instruction](#) exception.

The following instructions are considered as multi-cycle instructions in this context:

- Load operations
- LR instructions
- Multiply and Divide instructions
- Multi-cycle non-blocking extension instructions

To load a memory value into LP_COUNT, you must use an intermediate register as follows:

Example 5-1 Correct LP_COUNT Setup through a Register

```
LD    r1,[r0]          ; register loaded from memory
MOV   LP_COUNT, r1     ; LP_COUNT loaded from register
```

When reading the LP_COUNT register within a zero-overhead loop, the correct value is always returned. The LP_COUNT register can also be written at any point within the loop.

All writes to the LP_COUNT register take effect immediately after the writing instruction is complete, and after the loop-end mechanism detection has taken place. If an instruction writing to LP_COUNT is in the last position of a loop, any required change of program flow (that is, jump to [Loop Start Register, LP_START](#)) is completed before the LP_COUNT register is updated by that instruction.

As a result, writing LP_COUNT from the last instruction in the loop takes effect in the next loop iteration. Writing LP_COUNT from any other position in the loop takes effect in the current loop iteration.

5.1.10 Reserved Register (r61)

Register r61 is reserved. Any reference to core register r61 raises an [Illegal Instruction](#) exception.

5.1.11 Immediate Data Indicator, L IMM (r62)

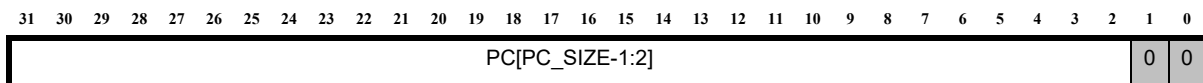
Register r62 is reserved for encoding long (32-bit) immediate (LIMM) data operands or to signify that the result of an instruction shall be discarded. Register r62 is therefore not available as a general purpose register.

When a LIMM is referenced as a source operand, it provides a 4-byte literal value which shall be located in the instruction address space in the 4 bytes following the instruction that references the LIMM. In this case the LIMM forms part of the referencing instruction.

When a LIMM is referenced in the context of a destination register operand, it serves as an indicator that no destination register is updated by that instruction. No actual LIMM data follows the instruction in this case.

5.1.12 Word-aligned Program Counter, PCL (r63)

Figure 5-2 PCL Register



Register r63 (PCL) is a read-only 32-bit value word-aligned Program Counter. r63 is used as a source operand in all instructions supporting PC-relative addressing. Bits [1:0] always return 0.

When read, the PCL register returns the address of the 32-bit word in which the current instruction begins. In contrast, the PC (see [Program Counter, PC](#)) auxiliary register returns the actual address of the committing instruction.

$PCL = PC \ \& \ 0xFFFF \ FFFC$

Any attempt to write to PCL as the destination register of an instruction raises an [Illegal Instruction](#) exception. If PC_SIZE is less than 32, the most significant 32-PC_SIZE bits of PCL are always read as zero.

5.2 Extension Core Registers

The register set is extensible in register positions 32-57 (r32-r57).

When MPY_OPTION > 6 or FPU_FMA_OPTION is configured, two extension core registers (r58 and r59) are included and comprise the 64-bit Accumulator (ACCH, ACCL).

In this manual, the ACC register refers to the register pair comprising the accumulator (ACCH, ACCL), according to the definition of a register pair given in [“64-Bit Data”](#) on page 123.

When a floating-point unit is included and the fused multiply-add instructions are enabled, the two extension core registers (r58 and r59) are included and comprise the third operand of each fused multiply-add or multiply-subtract instruction. When double-precision floating-point operations are configured, both ACCH and ACCL are used to represent the double-precision third operand as shown in [Figures 5-3 and 5-4](#). For single-precision fused multiply-add or multiply-subtract operations, the ACCL register is used to represent the third operand, as shown in [Figure 5-5](#):

Figure 5-3 ACCH for Double-Precision FPU operations

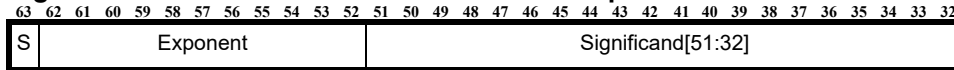


Figure 5-4 ACCL for Double-Precision FPU operations

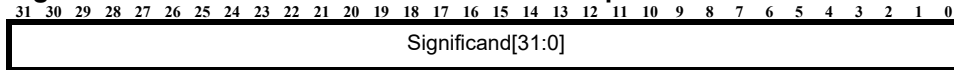
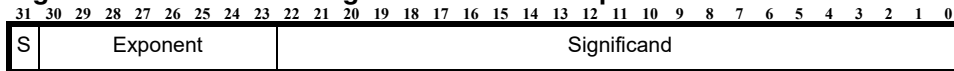


Figure 5-5 ACCL for Single-Precision FPU Operations



When a DSP unit with MAC is included in the processor, the two extension core registers (r58 and r59) are included and comprise the 64-bit Accumulator (ACCH, ACCL).

In this manual, the ACC register refers to the register pair comprising the accumulator (ACCH, ACCL), according to the definition of a register pair.



Note

The registers r58 and r59 are reserved and cannot be defined by APEX extensions. However, these registers are accessible (explicitly) by APEX instructions if the processor is configured to implement these registers.

If R58/R59 are used as a destination in a multiply/accumulate instruction, the update of the accumulator takes precedence over the write-back. In this case the write-back value is discarded.

5.2.1 Illegal Extension Core Register Usage

Any reference to a non-implemented core register raises an [Illegal Instruction](#) exception. See also, “[Illegal Core Register Usage](#)” on page [137](#).

5.3 Auxiliary Register Set

A small selection of auxiliary registers are baseline registers, and are therefore present in all ARCV2 systems. Other auxiliary registers are configurable and are present only if their defining extension option is included in the architectural configuration selected at build-time.

5.3.1 Baseline Auxiliary Registers

Table 5-5 lists the baseline set of auxiliary registers. The access code definitions of these registers are explained in Table 5-7.

Table 5-5 Baseline Auxiliary Register Set

Number	Auxiliary Register Name	Description
Current Architectural State		
0x4	Core Identity Register, IDENTITY	Processor identification register
0x6	Program Counter, PC	PC register (32-bit)
0x0A	Status Register, STATUS32	Status register (32-bit)
0x412	Branch Target Address, BTA	Branch target address
0x403	Exception Cause Register, ECR	Exception cause register
0x25	Interrupt Vector Base Register, INT_VECTOR_BASE	Interrupt vector base address
Saved Exception and Interrupt State		
0xD	Saved User Stack Pointer, AUX_USER_SP	Swap stack registers
0x400	Exception Return Address, ERET	Exception return address
0x401	Exception Return Branch Target Address, ERBTA	BTA saved on exception entry
0x402	Exception Return Status, ERSTATUS	STATUS32 saved on exception
0x404	Exception Fault Address, EFA	Exception fault address
Build Configuration Registers (BCR)		
0x60	Build Configuration Registers Version, BCR_VER	Build configuration registers version
0x63	BTA Configuration Register, BTA_LINK_BUILD	Build configuration for: BTA registers
0x68	Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD	Build configuration for: interrupts
0x6E	Core Register File Configuration Register, RF_BUILD	Build configuration for: core registers
0xC1	Instruction Set Configuration Register, ISA_CONFIG	Instruction set configuration BCR

5.3.2 Optional Architectural, Exception, and Interrupt State Auxiliary Registers

Table 5-6 lists the architectural, exception, and interrupt state-related auxiliary registers. These registers are available in the processor when the option `-sec_modes_option==true`.

Table 5-6 Auxiliary Register Set

Number	Auxiliary Register Name	Description
Current Architectural State		
0x9	Secure Status Register, SEC_STAT	Secure status register
0x26	Secure Interrupt Vector Base Register, INT_VECTOR_BASE_S	Secure interrupt vector base address
Saved Exception and Interrupt State		
0x38	Saved Normal Kernel Stack Pointer, AUX_KERNEL_SP	normal kernel stack swap register
0x39	Saved Secure User Stack Pointer, AUX_SEC_U_SP	Secure user stack swap register
0x3A	Saved Secure Kernel Stack Pointer, AUX_SEC_K_SP	Secure kernel stack swap register
0x3B	Saved Shadow Normal Stack Pointer, AUX_NSEC_SP	Saved shadow stack pointer register
0x268	Secure Jump and Link Indexed Top Address, NSC_TABLE_TOP	Secure Jump and link indexed region top address
0x269	Secure Jump and Link Indexed Base Address, NSC_TABLE_BASE	Secure Jump and link indexed region base address
0x406	Exception Secure Status Register, ERSEC_STAT	Exception secure status registers
0x407	Secure Exception Register, AUX_SEC_EXCEPT	Secure exception programming register
Build Configuration Registers (BCR)		
0x70	Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD	Build configuration for: secure interrupts

5.3.3 Register Access Permissions

Each table of auxiliary registers specifies the registers' auxiliary address, the name of the register, the register's LR and SR permissions, and a brief description of the purpose of the register. A key to the permission mnemonics is given in Table 5-7. All bullet entries “•” indicate cases where permission for the requested operation (read or write) is granted in the given operating mode (user, kernel). The Debug Host column indicates the read and write accessibility available for the Debug Host for each level of access rights.

**Note**

Access to auxiliary registers through side-effect, such as altering flags in the STATUS32 register, is governed by the privilege requirements of any instruction that makes an implicit read or write to an auxiliary register. [Table 5-7](#) refers only to the following:

- Reads through the LR instruction
- Writes through the SR instruction
- Reads or writes from the Debug host interface

In [Table 5-7](#), “P” indicates that any attempt to perform the requested operation raises a Privilege Violation exception; “I” indicates that the requested operation raises an Illegal Instruction exception; “-” indicates that the requested operation is ignored. In such cases, a read returns 0, and a write has no effect.

Table 5-7 Key to Auxiliary Register Access Permissions for LR and SR Instructions

Access	User Mode		Kernel Mode		Debug Host	
	Read	Write	Read	Write	Read	Write
r	•		•		•	-
R	P		•		•	-
w		•		•	-	•
rw	•	•	•	•	•	•
W		P		•	-	•
rW	•	P	•	•	•	•
RW	P	P	•	•	•	•
rG	•		•		•	•
RG	P		•		•	•
NN ^a	P ^b	P	P	P	-	-

a. NN indicates that the register is not accessible in the normal user, normal kernel, or the debug access in the normal mode.

b. Indicates that a EV_PrivilegeV exception is raised with ECR value 0x071020.

5.3.4 Optional Instruction Set Auxiliary Registers

When certain architectural configuration parameters are set at a level that is higher than their baseline level, additional auxiliary registers may be included. For example, when the `lpc_size` parameter is non-zero, the `LP_START` and, `LP_END` registers are included in the auxiliary register set.

Table 5-8 Zero Overhead Loop Auxiliary Registers (lpc_size > 0)

Address	Auxiliary register name	Description
0x2	Loop Start Register, LP_START	Loop start address (32-bit)
0x3	Loop End Register, LP_END	Loop end address (32-bit)

Similarly, when the `code_density_option` is `true`, the `JLI_BASE`, `LDI_BASE`, and `EI_BASE` registers are included because they provide an implicit operand to the `JLI_S`, `LDI_S`, and `EI_S` instructions respectively.

Table 5-9 Indexed Table Auxiliary Register

Address	Auxiliary Register Name	Description
0x290	Jump and Link Indexed Base Address, JLI_BASE	Jump and link indexed base address
0x291	Load Indexed Base Address, LDI_BASE	Load indexed base address
0x292	Execute Indexed Base Address, EI_BASE	Execute indexed base address

5.3.5 User Extension Auxiliary Registers

The ARCV2 architecture allows the pre-defined set of auxiliary registers to be extended with additional registers that are defined by user extensions. User Extensions define the meaning and accessibility of these extension auxiliary registers. In principle, the full range of 2^{32} auxiliary addresses are available for use as extension auxiliary register addresses, excluding the ones that are predefined by the ARCV2 architecture and any other Synopsys standard extensions and add-on components for ARCV2-based processor products.

The ARCV2 architecture can also be extended by the inclusion of additional user-defined instructions. Every user-defined extension instruction is associated with one of up to 32 extension groups. The XPU auxiliary register enables the access to these extension instruction groups in user mode. A build may contain one or more extensions. The user may control executing instructions of each extension in user mode by associating the extension with an XPU bit. The allocated XPU bit may be unique for each extension or shared between several extensions with the same user-mode privilege characteristics. This register is included only if there is at least one user-defined extension group in the build.



Note You cannot configure extensions with different modes (secure or normal) to use the same XPU bit; otherwise the XPU bit is read as zero and ignored on writes when accessed in the normal kernel mode.

Table 5-10 XPU, User Extension Permission Auxiliary Register

Number	Auxiliary Register Name	Description
0x410	User Mode Extension Enable Register, XPU	User extension permission register

When the ARCV2 architecture contains any user-defined extensions, the XFLAGS auxiliary register is included automatically in the build. This provides four additional flag bits that can be implicit inputs and outputs of extension instructions. For further details on the XFLAGS auxiliary register, see the description of the ALU Interface in the Databook for your ARCV2-based processor.

Table 5-11 Extension Flags Register, XFLAGS

Number	Auxiliary Register Name	Description
0x44F	User Extension Flags Register, XFLAGS	User extension flags register

5.3.6 Optional Build Configuration Registers

The ARCV2 architecture defines two special regions within the auxiliary address map. These are populated as required by the read-only registers which encode information about the build configuration of the system. These are located from addresses 0x60 to 0x7F and 0xC0 to 0xFF. Unused addresses in these series are all reserved.

5.3.7 Reset Value of Auxiliary Registers

The reset value of an auxiliary register is 0 unless it is explicitly stated in the description of the register.

5.3.8 Loop Start Register, LP_START

Address: 0x02

Access: rw

Reset: 0x0000_0000

Figure 5-6 LP_START Register (PC_SIZE == 32)

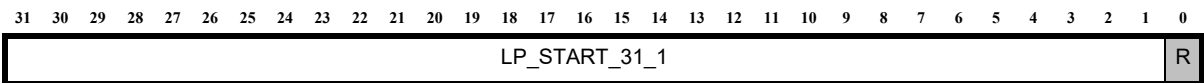
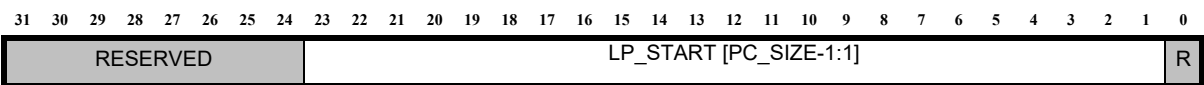


Figure 5-7 LP_START Register (PC_SIZE < 32)



The loop start (LP_START) register contains the address at which the current zero delay loop begins. [Figure 5-6](#) on [page 146](#) shows the format of this register. The loop start register can be set up with the LPcc instruction or can be manipulated using the auxiliary register access instructions (see [LR](#) and [SR](#)) format, and therefore, its size is determined by the PC_SIZE configuration parameter.

Programming cautions may exist when using the loop control registers. For more information, see [LP<cc>](#) instruction on [page 622](#).

Table 5-12 LP_START Field Description

Field	Bit	Reset	Description
LP_START_31_1	[31:1]	0	Address at which the current zero delay loop begins.
R0	[0]	0	Reserved. read-as-zero; ignored on write

5.3.9 Loop End Register, LP_END

Address: 0x03

Access: rw

Reset: 0x0000_0000

Figure 5-8 LP_END Register

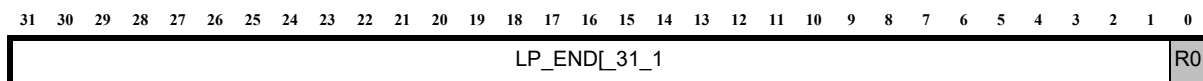
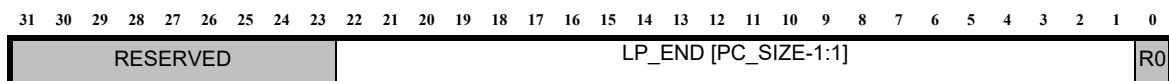


Figure 5-9 LP_END Register (PC_SIZE < 32)



The loop end (LP_END) register contains the address of the first instruction after the current zero delay loop. The loop end register can be set up with the LPcc instruction or can be manipulated using the auxiliary register access instructions (LR and SR) format, and therefore, its size is determined by the PC_SIZE configuration parameter.

Programming cautions may exist when using the loop control registers. For more information, see the LP<cc> instruction on [page 622](#) and “[Loop Count Register, LP_COUNT \(r60\)](#)” on [page 138](#).

Table 5-13 LP_END Field Description

Field	Bit	Reset	Description
LP_END_31_1	[31:1]	0	Address at which the current zero delay loop ends.
R0	[0]	0	Reserved. read-as-zero; ignored on write

5.3.10 Core Identity Register, IDENTITY

Address: 0x04

Access: r

Figure 5-10 IDENTITY Register

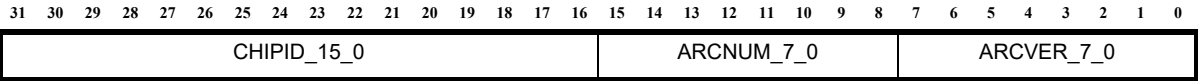


Figure 5-10 on page 148 shows the identity register (IDENTITY). This register contains the following fields:

Table 5-14 IDENTITY Field Descriptions

Field	Bit	Description
CHIPID_15_0	[31:16]	This field is deprecated. This field reads 0.
ARCNUM_7_0	[15:8]	This field allows you to read back the value of the arc_num[7:0] input pin on the core interface, set by the SoC integrator to uniquely identify each core in a multi-core system. Values: 0 to 255 Default: depends on the value on the input pin of the processor core. Caution: Cores in a multi-core system do not necessarily have sequential incrementing ARCNUM values starting from zero. Contact your silicon vendor to get details. Boot software can use this field to identify each processor core in a multi-core configuration and execute core-specific tasks. For example, software can load unique application code based on the core ID in a data-flow system, in which each core performs only a part of the algorithm.

Table 5-14 IDENTITY Field Descriptions (Continued)

Field	Bit	Description
ARCOVER_7_0	[7:0]	ARC baseline instruction set version number All non-enumerated values are reserved. ■ 0x41: ARC EM processor version 1.1a ■ 0x42: ARC EM processor version 3.0; added support for micro-DMA controller, CryptoPack extension, PDM interface, NV_ICCM, wide ICCM and DCCM, unaligned load and store. See the Release Notes for more information. ■ 0x43: ARC EM processor version 4.0 and ARC SEM version 1.0; Support for the address and data bus scrambling for caches, secure kernel and user modes, side-channel protection, safety island, secure debug, micro-DMA enhancements, and so on. See the Release Notes for details. ■ 0x44: ARC EM processor version 5.xxa and ARC SEM version 2.xxa that includes support for the non-maskable imprecise exception and updated EFA behavior (unlike previous releases, EFA now always exists in the processor). ■ 0x45: ARC EM Processor version that includes support for the TASK_SWITCH component and Overlay Manager. ■ 0x50: ARC HS processor version 1.0; first release of the ARC HS processor. ■ 0x51: ARC HS version 2.0; added multi-core support, added support for L2 cache, MMU support for multiple page sizes, increased the size of the supported CCMs, added support for up to eight register banks, added support for performance counters, ■ 0x52: ARC HS version 2.1; added single-cycle option for DCCM and data cache, dropped support for BVCI ■ 0x53: ARC HS version 3.0 and Embedded Vision processor (EV6x); support for cluster shared memory, Vector DSP ISA, APEX enhancements to improve performance and usability, added the DBNZ instruction, added PDM (Power Domain Management) unit to support external PMU, added FastMath extension pack as an option. See the Release Notes for details. ■ 0x54: ARC HS version 3.10a, 3.50a, 3.60a, and 4.00a, ARC VPX5 1.00a, EV6x (2.20a) and EV7x (3.00a) processors: support for shared memory configuration of CSM and L2 cache, unified DMI port, support for two peripheral ports. ■ 0x60 to 0xFF = Reserved

5.3.11 Program Counter, PC

Address: 0x06

Access: rG

Reset: 0x0000_0000

Figure 5-11 PC Register Addressing Full 32-bit Address Space

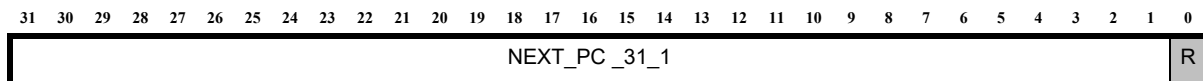
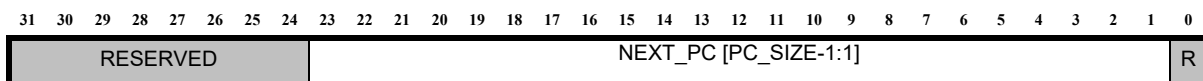


Figure 5-12 PC Register Addressing a Reduced Address Space (PC_SIZE == 32)



The PC register contains the address of the next instruction to be committed. Because all instructions are aligned to 16-bit half-word boundaries in memory, Bit 0 of PC is always zero. Any attempt to set Bit 0 of PC to 1 is ignored. When an LR instruction reads the PC register, it returns the address of the LR instruction itself.

The PC width can be configured to be less than 32 bits. In such case, all PC calculations are performed modulo the addressable range, causing the program counter to wrap around.

Table 5-15 lists the addressable range for different PC width configurations.

Table 5-15 ARCV2 Addressable Range vs. PC Size

PC Width (bits)	Addressable Range
16	64 KB
20	1 MB
24	16 MB
28	256 MB
32	4 GB

Table 5-16 PC Field Description

Field	Bit	Reset	Description
NEXT_PC__31_1_	[31:1]	0	Address (MSB 31 bits) of the next instruction to be committed

Table 5-16 PC Field Description

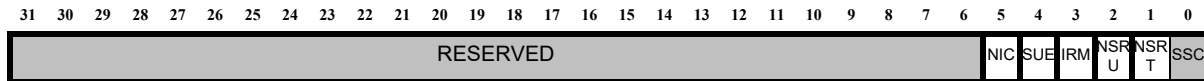
Field	Bit	Reset	Description
R0	[0]	0	Reserved. read-as-zero; ignored on write

5.3.12 Secure Status Register, SEC_STAT

Address: 0x09

Access: See [Table 5-17](#), [Table 5-18](#), and [Table 5-19](#)

Figure 5-13 SEC_STAT



This register is present only if `-sec_modes_option==true`.

The SFLAG instruction can modify the contents of the SEC_STAT register. This register cannot be written by a SR/AEX issued by processor in any operating mode; using any instruction other than SFLAG to access the SEC_STAT register causes an illegal instruction exception (ECR = 0x020000). However, a debug host can write the SEC_STAT register through the debug interface using a debug SR instruction.

[Table 5-17](#) lists the access permissions the SEC_STAT register bits when the core is operating in the normal mode and the secure mode.



Note

Fields marked as RAZ/IOW return 0 when read and ignore any writes.

Table 5-17 SEC_STAT Bit-Field Definitions and Read / Write Accessibility

Field	Bit	Normal User mode		Normal Kernel mode		Secure User mode		Secure Kernel mode	
		Read	Write	Read	Write	Read	Write	Read	Write
SSC	0	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	Yes
NSRT	1	RAZ	IOW	RAZ	IOW	Yes	Yes	Yes	Yes
NSRU	2	RAZ	IOW	RAZ	IOW	Yes	Yes	Yes	Yes
IRM	3	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	Yes
SUE	4	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	IOW
NIC	5	RAZ	IOW	Yes	IOW	RAZ	IOW	Yes	Yes

[Table 5-18](#) lists the debug access to the SEC_STAT register when the core is operating in the secure mode.

Table 5-18 SEC_STAT Bit Debug Access

Field	Bit	Debug port is Locked		Debug port is Unlocked in normal Mode		Debug port is unlocked in Secure Mode	
		Read	Write	Read	Write	Read	Write
SSC	0	RAZ	IOW	RAZ	IOW	Yes	Yes
NSRT	1	RAZ	IOW	RAZ	IOW	Yes	Yes
NSRU	2	RAZ	IOW	RAZ	IOW	Yes	Yes
IRM	3	RAZ	IOW	RAZ	IOW	Yes	Yes
SUE	4	RAZ	IOW	RAZ	IOW	Yes	Yes
NIC	5	RAZ	IOW	Yes	Yes	Yes	Yes

Table 5-19 SEC_STAT Field Description

Field	Bits	Description
SSC	[0]	The SSC bit is used to enable stack exceptions in the secure mode. When you include stack checking in the build options, you can read and write the STATUS32.SSC bit in the secure kernel mode and is read as zero and ignore on writes in the other modes. The following auxiliary registers are used for stack checking in the processor: ■ Secure Stack Region Top Address, S_STACK_TOP ■ Secure Stack Region Base Address, S_STACK_BASE The SSC bit is cleared on exception entry and restored on exception exit when the SEC_STAT register is restored. The SSC bit is unchanged on interrupt entry.
NSRT	[1]	Indicates whether a function return or an interrupt return can switch a normal operating mode to a secure operating mode. ■ 0x0: A function return is allowed to switch operating mode from normal to a secure operating mode. ■ 0x1: An interrupt return can switch the operating mode from normal to a secure operating mode. When the core returns from a normal mode to a secure mode through a function call or an interrupt return, and the return type is a mismatch with the mode defined by this NSRT bit, an exception is raised. This NSRT bit can be read in the secure mode only, and is read as zero in the normal mode. For debugger access, only secure mode unlock can access this bit.

Table 5-19 SEC_STAT Field Description (Continued)

Field	Bits	Description
NSRU	[2]	Indicates the operating mode that the core must return to when switching from a normal mode to the secure mode. ■ 0x0: Return from the normal mode to the secure kernel mode. ■ 0x1: Return to the normal mode to the secure user mode. When the core returns from a normal mode to a secure mode through a function call or an interrupt return, and the returning secure mode is a mismatch with the mode defined by this NSRU bit, an exception is raised. This NSRU bit can be read in the secure mode only, and is read as zero in the normal mode. For debugger access, only secure mode unlock can access this bit.
IRM	[3]	This bit determines the operating mode to which the processor must return to after executing an RTIE instruction from an interrupt. When returning from an interrupt, the processor operating mode switches to the IRM mode. If the operating mode of the instruction (the instruction is fetched from the secure or normal memory region) is different from the IRM mode, an EV_PrivilegeV exception is raised. This bit can be read only in the secure kernel mode and is read as zero in any other mode. For debugger access, this bit is accessible by debugger only if the debug port is unlocked in the secure mode.
SUE	[4]	This bit indicates whether a mode changing micro-op sequence is pending while a delay slot instruction is executing. It is used to determine that a micro-op sequence should execute after a breakpoint, sleep, or exception is handled after a branch has committed but before the branch-related delay slot instruction has committed. This bit is visible for processor read in the secure kernel mode and is read as zero for processor reads in other modes. Writes of 1 are ignored.
NIC	[5]	This enable bit allows the secure kernel mode code to control whether the SETI & CLRI instructions are available in normal mode, and whether the SLEEP & WEVT instructions can set the interrupt threshold when executed in the normal mode. This bit can be read in both secure kernel mode and normal kernel mode, and is read as zero in the normal or secure user modes.

5.3.13 Status Register, STATUS32

Address: 0x0A

Access: See the bit description for the bit-specific access information.

Figure 5-14 STATUS32 Register

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
IE	RESERVED										S	US	AD	RB[2:0]		ES	SC	DZ	L	Z	N	C	V	U	DE	AE	E[3:0]			H	

The status register, STATUS32 performs the following functions:

- Enables or disables certain actions within the processor.
- Contains a number of flags to indicate the status resulting from the following:
 - The evaluation of instructions.
 - The taking of interrupts.
 - The raising of exceptions.

The status register is therefore updated automatically by the processor during program execution. However, the FLAG, KFLAG, RTIE, SETI, and CLRI instructions can modify the content of the STATUS32 register.

The STATUS32 register cannot be written by a regular SR instruction regardless of the operating mode. However, a debug host can write to this register through the debug interface using a special debug SR instruction.

When the ARCV2-based processor reads the status register in user mode, only the Z, N, C, and V bits are visible; all other bits read as zero (RAZ). In kernel mode, all bits of the STATUS32 register are visible. When the processor writes the status register in user mode, only the Z, N, C, and V bits are modified; all other bits are ignored on write (IOW). In kernel mode, in addition to the Z, N, C, and V bits, the US, H, SC, AD, and DZ bits of the STATUS32 register can be modified using the FLAG instruction. A privileged version of the FLAG instruction, called KFLAG, also allows a kernel-mode process to modify the IE, AE, and RB bits.

The following tables lists the individual bits within STATUS32. Column 1 gives the field name, column 2 gives its bit-position within the register, and column 3 describes its purpose. Columns 4, 5, and 6 indicate the behavior of each bit in STATUS32 when read by an LR instruction or written by a FLAG instruction, executing in either user or kernel modes.

STATUS32 Behavior on Interrupt Entry or Exit

On interrupt entry, the STATUS32 register is either pushed to the stack or copied to STATUS32_P0, depending on FIRQ_OPTION and the level of the interrupt (P0 or otherwise). On exception entry, the STATUS32 register is copied to ERSTATUS. However, an EV_Trap exception clears the ES and DE fields before copying the STATUS32 register to [Exception Return Status, ERSTATUS](#), as EV_Trap is a post-commit

exception. The DE, ES, and DZ bits are all cleared when an interrupt or exception is taken, after the STATUS32 register has been saved.

Table 5-20 STATUS32 Bit-Field Definitions, Read / Write Accessibility

Field	Bit	Description	User mode		Kernel mode		Debug access	
			Read	Write	Read	Write	Read	Write
H	0	Halt flag	RAZ	IOW	Yes	Yes	Yes	Yes
E [3:0]	4 to 1	Interrupt priority operating level of the processor	RAZ	IOW	Yes	Yes	Yes	Yes
AE	5	Processor is in an exception state	RAZ	IOW	Yes	Yes	Yes	Yes
DE	6	Delayed branch is pending	RAZ	IOW	Yes	IOW	Yes	Yes
U	7	User mode	RAZ	IOW	Yes	IOW	Yes	Yes
V	8	Overflow status flag	Yes	Yes	Yes	Yes	Yes	Yes
C	9	Carry status flag	Yes	Yes	Yes	Yes	Yes	Yes
N	10	Negative status flag	Yes	Yes	Yes	Yes	Yes	Yes
Z	11	Zero status flag	Yes	Yes	Yes	Yes	Yes	Yes
L	12	Zero-overhead loop disable	RAZ	IOW	Yes	IOW	Yes	Yes
DZ	13	EV_DivZero exception enable	RAZ	IOW	Yes	Yes	Yes	Yes
SC	14	Enable stack checking	RAZ	IOW	Yes	Yes	Yes	Yes
ES	15	EI_S table instruction pending	RAZ	IOW	Yes	IOW	Yes	Yes
RB[2:0]	18 to 16	Select a register bank	RAZ	IOW	Yes	Yes	Yes	Yes
AD	19	Disable alignment checking	RAZ	IOW	Yes	Yes	Yes	Yes
US	20	User sleep mode enable	RAZ	IOW	Yes	Yes	Yes	Yes
IE	31	Interrupt Enable; enables interrupts at or above the priority level set in STATUS32.E	RAZ	IOW	Yes	Yes	Yes	Yes

Table 5-21 STATUS32 Bit-Field Definitions and Read / Write Accessibility when Secure Mode is Configured

Field	Bit	Description	Normal User mode		Normal Kernel mode		Secure User		Secure Kernel	
			Read	Write	Read	Write	Read	Write	Read	Write
H	0	Halt flag	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	Yes
E [3:0]	4 to 1	Interrupt priority operating level of the processor	RAZ	IOW	Yes	Yes if SEC_ST AT.NIC= =0	RAZ	IOW	Yes	Yes
AE	5	Processor is in an exception state	RAZ	IOW	Yes	Yes	RAZ	IOW	Yes	Yes
DE	6	Delayed branch is pending	RAZ	IOW	Yes	IOW	RAZ	IOW	Yes	IOW
U	7	User mode	RAZ	IOW	Yes	IOW	RAZ	IOW	Yes	IOW
V	8	Overflow status flag	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
C	9	Carry status flag	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
N	10	Negative status flag	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Z	11	Zero status flag	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
L	12	Zero-overhead loop disable	RAZ	IOW	Yes	IOW	RAZ	IOW	Yes	IOW
DZ	13	EV_DivZero exception enable	RAZ	IOW	Yes	Yes	RAZ	IOW	Yes	Yes
SC	14	Enable stack checking	RAZ	IOW	Yes	Yes	RAZ	IOW	Yes	Yes
ES	15	EI_S table instruction pending	RAZ	IOW	Yes	IOW	RAZ	IOW	Yes	IOW
RB[2:0]	18 to 16	Select a register bank	RAZ	IOW	RAZ	IOW	RAZ	IOW	RAZ	IOW
AD	19	Disable alignment checking	RAZ	IOW	RAZ	IOW	RAZ	IOW	RAZ	IOW
US	20	User sleep mode enable	RAZ	IOW	Yes	Yes	RAZ	IOW	Yes	Yes
S	21	Secure mode	RAZ	IOW	RAZ	IOW	Yes	IOW	Yes	IOW
IE	31	Interrupt Enable; enables interrupts at or above the priority level set in STATUS32.E	RAZ	IOW	Yes	Yes if SEC_ST AT.NIC= =0	RAZ	IOW	Yes	Yes

Table 5-22 STATUS32 Bit-Field Definitions and Read / Write Accessibility in Secure Debug Mode

Field	Bit	Locked Debug		Debug Port unlocked in normal Mode		Debug Port unlocked in Secure Mode		Locked Debug		Debug Port unlocked in normal Mode		Debug Port unlocked in Secure Mode	
		Core is operating in the normal mode						Core is operating in the secure mode					
		Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
H	0	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	Yes	Yes	Yes
E [3:0]	4 to 1	RAZ	IOW	Yes	Yes if SEC_ST AT.NIC= =0	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
AE	5	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
DE	6	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
U	7	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
V	8	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
C	9	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
N	10	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
Z	11	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
L	12	RAZ	IOW	Yes	IOW	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
DZ	13	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
SC	14	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	Yes	Yes	Yes
ES	15	RAZ	IOW	Yes	IOW	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes
RB[2:0]	18 to 16	RAZ	IOW	RAZ	IOW	RAZ	IOW	RAZ	IOW	RAZ	IOW	RAZ	IOW
AD	19	RAZ	IOW	RAZ	IOW	RAZ	IOW	RAZ	IOW	RAZ	IOW	RAZ	IOW
US	20	RAZ	IOW	Yes	Yes	Yes	Yes	RAZ	IOW	Yes	Yes	Yes	Yes
S	21	RAZ	IOW	Yes	IOW	Yes	IOW	RAZ	IOW	Yes	IOW	Yes	IOW
IE	31	RAZ	IOW	Yes	Yes if SEC_ST AT.NIC= =0	Yes	Yes	RAZ	IOW	Yes	IOW	Yes	Yes

IE

The IE bit indicates whether interrupts are enabled in the processor. If IE is set to 0, the E[3:0] bits are ignored. If IE bit is set 1, interrupts are enabled and only those interrupts at a higher priority or equal to E[3:0] are enabled.

When an ARCV2 processor is configured without interrupts (`HAS_INTERRUPTS = 0`), the `STATUS32.IE` and `STATUS32.E` fields are read as zero and ignored on write. The `SETI` and `CLRI` instructions do not modify `STATUS32` bits in this case, as these instructions raise an [Illegal Instruction](#) exception.

S

The `S` bit indicates the operating mode of the core for the current instruction accessing the `STATUS32` register. The current mode (secure or normal) of the core is controlled by the MPU region match for the instruction fetch interface. If an instruction is being executed from a region that is marked as secure, then the core is in secure mode and the `S` bit is set to 1. If the instruction accessing the `STATUS32` register is executed from a normal region, then the core is in normal mode and the `S` bit is set to 0. This operating mode combined with the `U` bit determines whether the access was in a secure kernel mode, secure user mode, normal user mode, or normal kernel mode.

The `S` bit is supported only for ARCV2-based processors when `-sec_modes_option==true`.

US

This bit controls the behavior of the `WLFC`, if configured using the `ATOMIC_OPTION` instruction set configuration option, and `WEVT` instructions in user mode.

When `US==0`:

- In user mode, the `WLFC` instruction behaves as a `NOP`
- In user mode, the `WEVT` instruction raises a Privilege Violation

When `US==1`:

- The user-mode `WLFC` instruction behaves the same as a kernel-mode `WLFC` instruction
- The user-mode `WEVT` instruction behaves the same as a kernel-mode `WEVT` instruction

The `US` bit is read-as-zero (RAZ) when the `STATUS32` register is read in user mode. You modify the `STATUS32.US` bit in kernel mode using the `FLAG` and `KFLAG` instructions. This bit is also replicated in all copies of `STATUS32` that are taken when entering an exception or interrupt, such as `ERSTATUS`. The bit is restored when returning from exception or interrupt using the privileged `RTIE` instruction.

AD

The `AD` bit is present only in processors that support non-aligned data references. The `AD` bit indicates whether alignment checks on data memory references are disabled. If this bit is set to 1, all data memory alignment checks, except for the `EX`, `ENTER_S`, and `LEAVE_S` instructions, are disabled, and the processor does not raise `EV_Misaligned` ([Misaligned Data Access](#)) exceptions. This information is accessible to software by inspecting the `NON_ALIGNED` field in the [Instruction Set Configuration Register, ISA_CONFIG](#) register. If this feature is not supported, or if the `AD` bit is set to 0, natural alignment of all data references, as defined in “[Data Layout in Memory](#)” on page 122 are enforced by raising an `EV_Misaligned` exception ([Misaligned Data Access](#)) on all data references that are not aligned to the natural boundary of the data object being referenced.

The `AD` bit is set to 0 on reset.

Unaligned load and stores are not supported when `-sec_modes_option==true`.

RB_2_0_

Register banks are not supported in ARCV2-based processors with the secure features.

The RB field indicates the register bank that the processor is currently using. In kernel mode, use the KFLAG instruction to write to the STATUS32.RB[2:0] to select a register bank. Accesses to core registers from thereon are made to the registers in the selected register bank. The configuration parameter, RGF_NUM_BANKS, defines the number of banks in a processor. The value written to the STATUS32.RB field must be less than RGF_NUM_BANKS.



Note

If RGF_NUM_BANKS == 1, RB[2:0] is read as zero and ignored on write.

When FIRQ_OPTION is 1 and RGF_NUM_BANKS > 1, the interrupt entry sequence sets the register bank number STATUS32.RB to 1, the second register bank, when entering a priority P0 interrupt. Similarly, on P0 interrupt exit, the processor returns STATUS32.RB to its previous value by restoring STATUS32.RB from STATUS32_P0.

ES

When an EI_S (Execute Indexed) instruction is executed, the address of the next sequential instruction after EI_S is placed in the BTA register and the ES bit of the STATUS32 register is set to 1. The ES bit indicates that on commit, the next instruction performs an implicit branch to BTA. This action ensures that control returns to the position after the EI_S instruction after its target instruction has been executed. If the ES bit is set to 1 when an instruction completes, PC is set to the value in BTA and the ES bit is cleared.

SC

The SC bit is used to enable stack exceptions. When you include stack checking in the build options, you can read and write the STATUS32.SC bit in kernel mode. In user mode, the STATUS32.SC bit is read as zero and ignored on write. The following auxiliary registers are used for stack checking in the normal mode:

- User Stack Region Top Address, USTACK_TOP and User Stack Region Base Address, USTACK_BASE (KSTACK_TOP, USTACK_TOP), (USTACK_BASE, KSTACK_BASE)
- Stack Region Configuration Register, STACK_REGION_BUILD (STACK_REGION_BUILD BCR)

The SC bit is cleared on exception entry and restored on exception exit when the STATUS32 register is restored. The SC bit is unchanged on interrupt entry.

DZ

DZ indicates whether the EV_DivZero exception is enabled on division by zero. When DZ is set to 1, any attempt to execute a DIV, DIVU, REM, or REMU instruction, with a divisor of 0, raises an EV_DivZero exception. When DZ is set to 0, a division by zero does not raise an exception. The DZ bit is ignored on write and read as zero when there is no hardware divider configured, regardless of operating mode. The DZ bit is cleared on interrupt or exception entry, and is restored on return from an interrupt or exception.

L

L indicates whether the zero-overhead loop mechanism is disabled. L is set to 1, disabling zero-overhead loops, on an interrupt or exception. L is reset to its previous value when the STATUS32 register is restored to its previous state. L is also cleared when a loop instruction (LPcc) is executed.

U

U indicates user mode. User mode restricts access to privileged machine state and prevents execution of privileged instructions. When U is 0, kernel mode allows full access to all state and all instructions. Kernel mode is entered on [Reset](#), interrupts, or exceptions. U is reset to its previous value on interrupt or exception exit when the STATUS32 register is restored to its pre-exception or pre-interrupt state.

DE

When a branch or jump with a delay slot is executed, if its condition is true and the branch is taken, the target address is placed in the Branch Target Address (BTA) register and the DE bit of the STATUS32 register is set to 1. The DE bit indicates the presence of an outstanding delayed branch. Whenever an instruction completes successfully with the STATUS32 [DE] bit set to 1, the next PC value is always set to the value in the BTA register. If a conditional branch or jump with a delay slot is not taken, the DE bit is set to 0 and the BTA register is not updated. The DE bit is also set to 0 by execution of all non-branch instructions.

On an exception or interrupt return, the STATUS32 register is restored by the [RTIE](#) instruction. If the STATUS32[DE] bit is set true as a result of the RTIE operation, the Branch Target Address register ([Branch Target Address, BTA](#)) is simultaneously restored from the Exception Branch Target Address register ([Exception Return Branch Target Address, ERBTA](#)). The DE bit is only readable by an external debugger or from kernel mode.

AE

The AE bit is set on entry to an exception. Therefore, this bit indicates that an exception is active and that the Exception Return Address register (see [Exception Return Address, ERET](#)) is valid. When the return from interrupt or exception instruction ([RTIE](#)) is executed, AE is restored from the AE bit in the ERSTATUS register.

E[3:0]

The E[3:0] bits encode the interrupt priority threshold of the processor. Only interrupts at a higher priority or equal to E[3:0] are enabled. E[3:0] is located in STATUS32[4:1]. For example, set $E = 4$ to enable only interrupts at priority 4 (P4) or higher priority than P4 (P0, P1, P2, and P3). Similarly, set $E = 15$ to enable

interrupts at all priority levels. You can enable interrupts by setting the IE bit using the [SETI](#) instruction or when the [RTIE](#) instruction restores STATUS32. You can clear the IE bit using the [CLRI](#) instruction.

**Note**

For more information about interrupt priority levels, see [Exception and Interrupt Priority](#).

H

The H bit when set to 1 indicates that the core is halted. When `-sec_modes_option==true`, you can write to this using the FLAG instruction in the secure kernel mode only. If you attempt to set STATUS32.H to 1 in the normal kernel mode, a Privilege Violation is generated. Attempts to set the H bit using FLAG in both normal user mode and secure user mode are ignored.

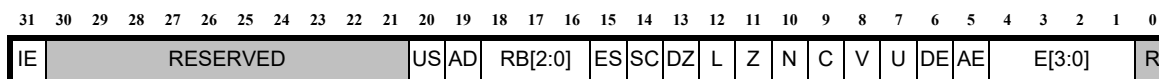
All fields, except the H bit, are set to 0 when the processor is [Reset](#). The H bit is set depending on the configuration of the processor run state on [Reset](#).

5.3.14 Status Register Priority 0, STATUS32_P0

Address: 0x0B

Access: RW

Figure 5-15 STATUS32_P0 Register



When fast interrupts are enabled in the processor, on the highest priority interrupt (P0) entry, the processor stores the value of the STATUS32 register to the STATUS32_P0 register.

In the STATUS32_P0 register, Bit 0 must always be set to zero. When read, Bit 0 returns zero.

On entry to P0 interrupt, if FIRQ_OPTION is enabled, the STATUS32 register is copied to STATUS32_P0. On return from the highest priority interrupt, P0, the STATUS32 register is restored from STATUS32_P0.

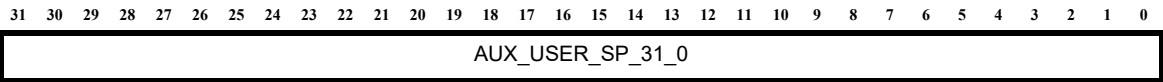
5.3.15 Saved User Stack Pointer, AUX_USER_SP

Address: 0x0D

Access: RW

Reset: 0x0000_0000

Figure 5-16 AUX_USER_SP Register



You can read and write to the AUX_USER_SP register in the normal and secure kernel modes; this register is protected and inaccessible in user mode. This register is used to save the normal user stack pointer when the core is switching modes from a normal user mode to the other modes. This instruction can also be used to efficiently exchange LP_START or LP_END with a core register during full context save and restore sequences.

Stack pointers can be swapped as follows with a single 32-bit instruction:

```
AEX R28, [0xD] ;; encoded with operand format b, u6
```

Table 5-23 AUX_USER_SP Field Description

Field	Bit	Reset	Description
AUX_USER_SP_31_0	[31:0]	0	Saved User Stack Pointer

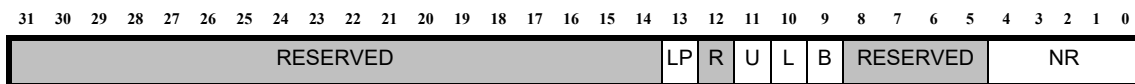
5.3.16 Interrupt Context Saving Control Register, AUX_IRQ_CTRL

Address: 0x0E

Access: When SecureShield 2+2 mode is not configured: RW
When SecureShield 2+2 mode is configured: RW in the secure mode and R in the normal mode

Reset: 0x0000_0000

Figure 5-17 AUX_IRQ_CTRL Register



The AUX_IRQ_CTRL register controls the behavior of automated register save and restore or prolog and epilog sequences during interrupt entry and exit, and context save and restore instructions.

This register is read as zero and ignored on write if fast interrupts are present and only one priority level (P0) is configured. A fast interrupt with only one priority level does not emit a prolog or epilog sequence, so you need not program the set of saved and restored registers. This register is present in a build only if the processor is configured to include interrupts.

When `-sec_modes_option==true`, read or write operations from the secure user mode causes privilege violation (0x070000). Read from the normal user mode causes privilege violation (0x070000). Writes from the normal mode causes illegal instruction violation (0x020000).

On reset, this register contains 0x00000000.

Table 5-24 AUX_IRQ_CTRL Field Description

Field	Bit	Reset	Description
NR	[4:0]	0	Indicates number of general-purpose register pairs saved, from 0 to 8/16. This register saturates at 16 when the <code>RGF_NUM_REGS == 32</code> , and 8 when the <code>RGF_NUM_REGS == 16</code> . The set of registers saved include the lowest numbered registers which are implemented in the register file. For 16 entry register files, the registers implemented and saved are not contiguous, see Table 5-3 .
RESERVED	[8:5]	0	Reserved. read-as-zero; ignored on write
B	[9]	0	Indicates whether to save and restore BLINK (ignored if NR is greater than or equal to 16.)
L	[10]	0	Indicates whether to save and restore loop registers (<code>LP_COUNT</code> , <code>LP_START</code> , <code>LP_END</code>)
U	[11]	0	Indicates if user context is saved to user stack
RESERVED	[12]	0	Reserved. read-as-zero; ignored on write

Table 5-24 AUX_IRQ_CTRL Field Description

Field	Bit	Reset	Description
LP	[13]	0	Indicates whether to save and restore code-density registers (EI_BASE, JLI_BASE, LDI_BASE)
RESERVED	[31:14]	0	Reserved. read-as-zero; ignored on write

5.3.17 Interrupt Vector Base Register, INT_VECTOR_BASE

Address: 0x25

Access: RW

Reset: Based on the ARCHitect configuration

Figure 5-18 INT_VECTOR_BASE Register (PC_SIZE == 32)

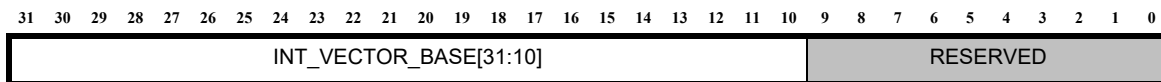
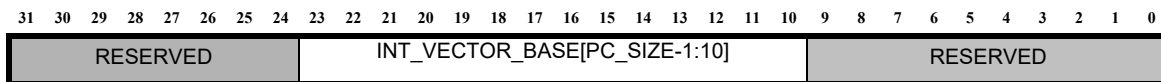


Figure 5-19 INT_VECTOR_BASE Register (PC_SIZE < 32)



The Interrupt Vector Base register (INT_VECTOR_BASE) contains the base address of the interrupt vectors. On **Reset**, the interrupt vector base address is loaded with a value from the interrupt system (see “[Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD](#)” on page 208). This reset value is configured at build time through the `-intvbase_preset` parameter and can be read from INT_VECTOR_BASE at any time. During program execution, the interrupt vector base can be changed by writing to INT_VECTOR_BASE. The interrupt vector base address can be set to any 1K-aligned address. The bottom 10 bits are ignored for writes and return 0 on reads.

The width of the Interrupt vector base field is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

When an MMU is configured, the INT_VECTOR_BASE register must be set to an address in the non-translated memory space. In ARC EM family of processors, this address is within the upper 2 GBytes of the memory space. This design avoids double-fault exceptions from occurring when accessing the vector table to service a first exception. Fetching the vectors is then not subject to TLB misses or other TLB exceptions.

If the `-intvbase_ext` option is set to `true`, on processor reset, this register stores the address specified on the external signal `intvbase_in` in the upper 22 bits of this register.

Table 5-25 INT_VECTOR_BASE Field Description

Field	Bit	Reset	Description
INT_VECTOR_BASE_31_10 —	[31:10]	ARCHitect configurable	Interrupt Vector Base MSB.
RESERVED	[9:0]	0	Reserved. read-as-zero; ignored on write

5.3.18 Secure Interrupt Vector Base Register, INT_VECTOR_BASE_S

Address: 0x26

Access: RW in the secure mode

Figure 5-20 INT_VECTOR_BASE_S Register (PC_SIZE == 32)

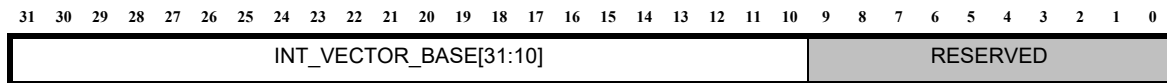
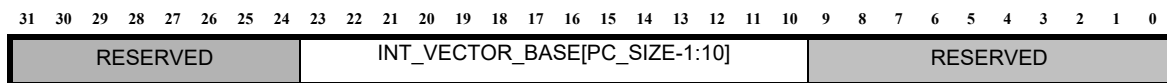


Figure 5-21 INT_VECTOR_BASE_S Register (PC_SIZE < 32)



This register exists in the processor only when `-sec_modes_option==true`. The secure interrupt vector base register (INT_VECTOR_BASE_S) contains the base address of the secure interrupt vectors. On [Reset](#), the interrupt vector base address is loaded with a value from the interrupt system (see [“Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD”](#) on page 209). This reset value is configured at build time through the `-intvbase_preset_s` parameter and can be read from INT_VECTOR_BASE_S at any time. During program execution, the interrupt vector base can be changed by writing to INT_VECTOR_BASE_S. The bottom 10 bits are ignored for writes and return 0 on reads.

The width of the Interrupt vector base field is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

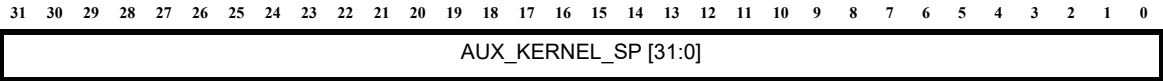
The base address for the secure interrupt table must be 1K-byte aligned. This address must be in a secure memory region as defined by the MPU. If the vector address for a secure interrupt or secure exception is in the normal memory region or the vector address for a normal interrupt or exception is in the secure memory region, an EV_Machine Check exception is triggered. If this EV_Machine Check exception happens during the servicing of an exception, a double fault is generated. Further if this double fault exception is triggered from an normal mode exception, the exceptions use special ECRs to indicate these fault conditions. If a further violation is triggered on the double fault exception or handler, a triple fault occurs and the core is halted.

5.3.19 Saved Normal Kernel Stack Pointer, AUX_KERNEL_SP

Address: 0x38

Access: RW in the secure mode

Figure 5-22 AUX_KERNEL_SP Register



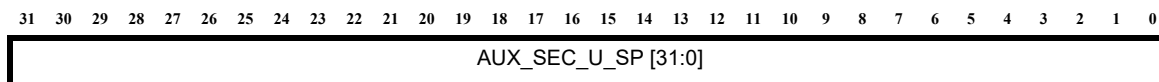
This register exists in the processor only when `-sec_modes_option==true`. You can read and write to the AUX_KERNEL_SP register in the normal and secure kernel modes; this register is protected and inaccessible in user mode. This register is used to save the normal kernel stack pointer when the core is switching modes from a normal kernel mode to the other modes.

5.3.20 Saved Secure User Stack Pointer, AUX_SEC_U_SP

Address: 0x39

Access: RW in the secure modes

Figure 5-23 AUX_SEC_U_SP Register



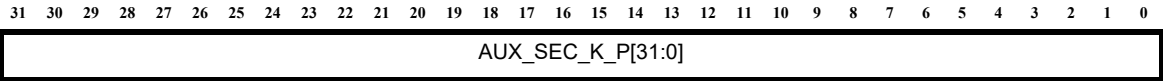
This register exists in the processor only when `-sec_modes_option==true`. You can read and write to the AUX_SEC_U_SP register in the secure kernel modes; this register is protected and inaccessible in user mode. This register is used to save the secure user stack pointer when the core is switching modes from a secure user mode to the other modes.

5.3.21 Saved Secure Kernel Stack Pointer, AUX_SEC_K_SP

Address: 0x3A

Access: RW in the secure modes

Figure 5-24 AUX_SEC_K_SP Register



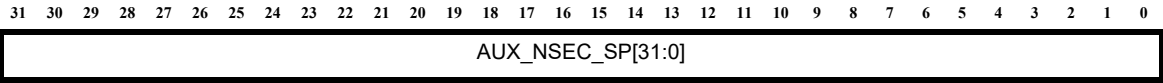
This register exists in the processor only when `-sec_modes_option==true`. You can read and write to the AUX_SEC_K_SP register in the secure kernel modes; this register is protected and inaccessible in user mode. This register is used to save the secure kernel stack pointer when the core is switching modes from a secure kernel mode to the other modes.

5.3.22 Saved Shadow Normal Stack Pointer, AUX_NSEC_SP

Address: 0x3B

Access: r in the secure modes

Figure 5-25 AUX_NSEC_SP Register



This register exists in the processor only when `-sec_modes_option==true`. This register is used to read the normal user and kernel stack pointer. When `STATUS32.U==1`, this register returns the normal user mode stack pointer, `AUX_USER_SP`. When `STATUS32.U==0`, this register returns the normal kernel mode stack pointer, `AUX_KERNEL_SP`.

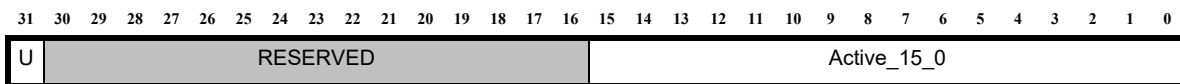
5.3.23 Active Interrupts Register, AUX_IRQ_ACT

Address: 0x43

Access: When SecureShield 2+2 mode is not configured: RW
When SecureShield 2+2 mode is configured: RW in the secure mode and r in the normal mode

Reset: 0x0000_0000

Figure 5-26 AUX_IRQ_ACT Register



This register records the current stack of nested interrupt handlers. When you set the least significant bit to 1 in the Active field, it indicates that the highest priority interrupt is active interrupt. For more information about interrupt prioritization and preemption, see [“Interrupt Prioritization and Preemption”](#) on page 236.

The U bit records whether the processor was in user or kernel mode when the outermost handler was entered. This register determines whether to restore the user or kernel stack pointer before returning from the outermost interrupt handler.

When `-sec_modes_option==true`, read or write operations from the secure user mode causes privilege violation (0x070000). Read from the normal user mode causes privilege violation (0x070000). Writes from the normal mode causes illegal instruction violation (0x020000).

This register is present in a build only if the processor is configured to include interrupts. On reset, this register contains 0x00000000.

Table 5-26 AUX_IRQ_ACT Field Description

Field	Bit	Reset	Description
Active_15_0	[15:0]	0	Active Interrupt Priority level Bits corresponding to the secure interrupts are read as zero when read from a normal kernel mode.
Reserved	[30:16]	0	Reserved. Read-as-zero; ignored on write.
U	[31]	0	Snapshot of the STATUS32.U bit when an interrupt is taken at a point where Active[15:0] == 0. If the outermost interrupt is a secure interrupt, read access on this bit from normal kernel mode is read as zero.

5.3.24 Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING

Address: 0x200

Access: R

Reset: 0x0000_0000

Figure 5-27 IRQ_PRIORITY_PENDING Register



This register provides software visibility of all priority levels at which an interrupt is pending, including levels with lower priority than the currently-active interrupt.

When `-sec_modes_option==true`, if a bit in the IRQ_PRIORITY_PENDING register is set to 1, and all the corresponding pending interrupts are secure interrupts, a read to this register in the normal kernel mode returns zeros. If some or all of the pending interrupts are normal interrupts, a read to this register returns 1's in the corresponding normal interrupt bits.

To know the pending interrupts (secure or normal) of all priority levels, read this register in the secure kernel mode.

The width of this register is determined by the number of interrupt priority levels configured in the system. This register is present in a build only if the processor is configured to include interrupts.

On reset, this register contains 0x00000000 for configured interrupts.

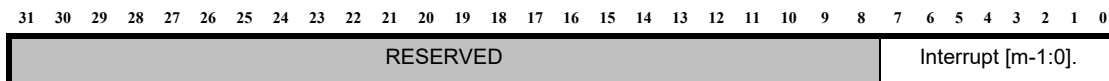
Table 5-27 IRQ_PRIORITY_PENDING Field Description

Field	Bit	Reset	Description
PENDING_N_1_0	[N-1:0] Where N is the number of interrupt priority levels configured.	0	Bit i indicates whether there is a pending normal interrupt at priority level i. If fewer than 16 priority levels are configured, the unused bits are read as zero.
Reserved	[31:N]	0	Reserved. Read-as-zero; ignored on write.

5.3.25 Software Interrupt Trigger, AUX_IRQ_HINT

Address: 0x201
 Access: RW
 Reset: 0x0000_0000

Figure 5-28 AUX_IRQ_HINT Register



In addition to the SWI instruction, the interrupt system allows software to generate a specific interrupt by writing to the software interrupt trigger register (AUX_IRQ_HINT). All interrupts can be generated through the AUX_IRQ_HINT register. The AUX_IRQ_HINT register can be written through ARCV2-based code or from the host (see [“The Host”](#) on page 779).

When `-sec_modes_option==true`, when you write to this register in the normal mode, you can only generate the normal interrupts. Attempts to generate secure interrupts from the normal interrupts are ignored. You can generate secure interrupts from the secure kernel mode only.

The software triggered interrupt mechanism can be used even if there are no associated interrupts connected to the ARCV2-based processor.

Writing the chosen interrupt value to the AUX_IRQ_HINT register generates a software triggered interrupt.



Note

The AUX_IRQ_HINT register uses only Bits [m-1:0] to determine the interrupt number.

Writing a value of any unimplemented interrupt, such as 0, clears any software triggered interrupt.

A read from the AUX_IRQ_HINT register returns the value of the current software triggered interrupt. A new interrupt must not be generated using the software triggered interrupt system until any outstanding interrupts have been serviced. The AUX_IRQ_HINT register must be read and checked as 0x0 before a new value is written.

Use the AUX_IRQ_HINT register to set the associated interrupt before generating a pulse sensitive interrupts.

The width of the interrupt fields is determined by the number of interrupts configured in the system. If M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$, where `ceil` is the function to round to the next higher integer.

5.3.26 Interrupt Priority Register, IRQ_PRIORITY

Address: 0x206

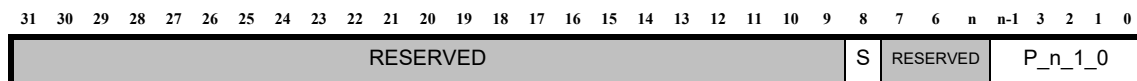
Access: RW

When SecureShield 2+2 mode is not configured: RW

When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x0000_0000

Figure 5-29 IRQ_PRIORITY Register



This banked register allows software to set and examine the current priority level of the interrupt selected by the [Interrupt Select, IRQ_SELECT](#) register.

A unique copy of IRQ_PRIORITY exists for each interrupt in the processor.



Note

If reserved bits are programmed to non-zero or the interrupt priority set is to $> N-1$, the priority is clamped to $N - 1$.

The width of the priority field is determined by the number of interrupt priority levels configured in the system. If N is the number of interrupt priority levels configured, $n = \text{ceil}(\log_2(N))$, where ceil is the function to round to the next higher integer. You can configure up to 16 priority levels. This register is present in a build only if the processor is configured to include interrupts.

- The external interrupts have a default priority level of zero (0).
- If performance counters use an interrupt,
 - the default priority is one (1) if the performance counter interrupts are configured to use more than one priority level.
 - the default priority is zero (0) if the performance counter interrupts are configured to use only one priority level.

Table 5-28 IRQ_PRIORITY Field Description

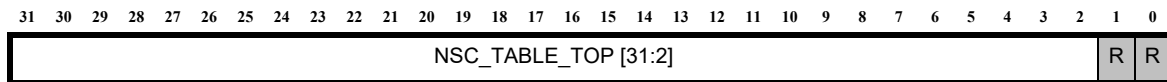
Field	Bit	Reset	Description
P_n_1_0	[n-1:0] If N is the number of interrupt priority levels configured, $n = \text{ceil}(\log_2(N))$	0	Value of IRQ_PRIORITY, for the selected interrupt. When S==1, this bit is read as zero and ignored on writes in the normal mode.
S	[8]	0	Indicates if the interrupt is a secure interrupt or normal interrupt. This bit is available only if -sec_mode_options==true. 0x0: generates a normal interrupt. 0x1: generates a secure interrupt. The timer 0 and timer 1 interrupts are by default set as normal interrupts (S==0x0). Other interrupts are by default assumed to be secure interrupts (S==1). This bit is read as zero and ignored on writes in the normal mode.
Reserved	[31:9]	0	Reserved. Read-as-zero; ignored on write.

5.3.27 Secure Jump and Link Indexed Top Address, NSC_TABLE_TOP

Address: 0x268

Access: RW in the secure mode

Figure 5-30 NSC_TABLE_TOP Register when PC_SIZE == 32



This register exists in the processor only when `-sec_modes_option==true`. This register contains the top address in the secure memory region, at which the jump table used by SJLI instructions resides.

Secure mode functions can be registered as normal callable at compile time. This causes an entry to be added to a special jump table (defined by the range: NSC_TABLE_BASE and NSC_TABLE_TOP) in the secure memory region containing the start address of that function. The call is then implemented in normal code using a new indexed-jump & link instruction (SJLI) that references the relevant offset in the special jump table.

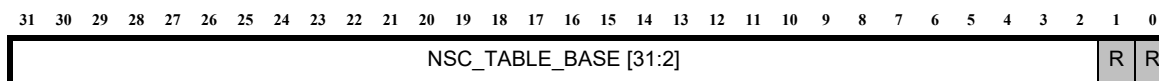
The width of this register is determined by the PC size. If `pc_size` is 16 or 24, the unused MSB bits of NSC_TABLE_BASE/NSC_TABLE_TOP are read as zero and ignored on writes.

5.3.28 Secure Jump and Link Indexed Base Address, NSC_TABLE_BASE

Address: 0x269

Access: RW in the secure mode

Figure 5-31 NSC_TABLE_BASE Register when PC_SIZE == 32



This register exists in the processor only when `-sec_modes_option==true`. This register contains the base address in the secure memory region, at which the jump table used by SJLI instructions resides.

Secure mode functions can be registered as normal callable at compile time. This causes an entry to be added to a special jump table (defined by the range: NSC_TABLE_BASE and NSC_TABLE_TOP) in the secure memory region containing the start address of that function. The call is then implemented in normal code using a new indexed-jump & link instruction (SJLI) that references the relevant offset in the special jump table.

The width of this register is determined by the PC size. If pc_size is 16 or 24, the unused MSB bits of NSC_TABLE_BASE/NSC_TABLE_TOP are read as zero and ignored on writes.

5.3.29 Jump and Link Indexed Base Address, JLI_BASE

Address: 0x290

Access: rw

Reset 0x0000_0000

Figure 5-32 JLI_BASE Register when PC_SIZE == 32

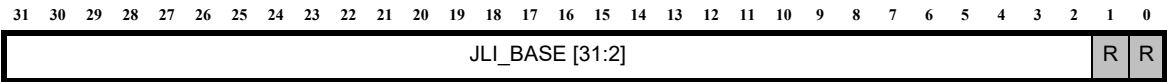
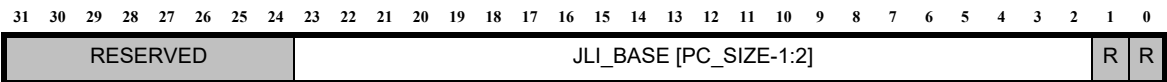


Figure 5-33 JLI_BASE Register when PC_SIZE < 32



The JLI_BASE register contains a base address in program memory, at which the jump table used by JLI_S instructions resides.

The width of the JLI_BASE register is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

5.3.30 Load Indexed Base Address, LDI_BASE

Address: 0x291

Access: rw

Reset 0x0000_0000

Figure 5-34 LDI_BASE Register when ADDR_SIZE == 32

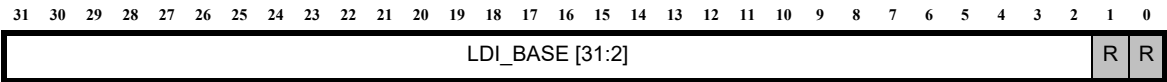
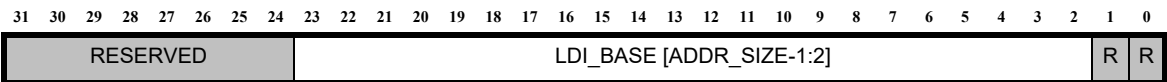


Figure 5-35 LDI_BASE Register when ADDR_SIZE < 32



The LDI_BASE register contains a base address in data memory, at which the constant pool used by LDI_S instruction resides.

The width of the LDI_BASE register is determined by the address size. Reading any reserved bits returns 0 and writes to such bits have no effect.

5.3.31 Execute Indexed Base Address, EI_BASE

Address: 0x292

Access: rw

Reset 0x0000_0000

Figure 5-36 EI_BASE Register when PC_SIZE == 32

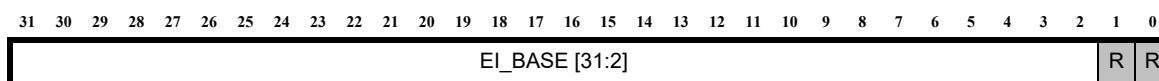
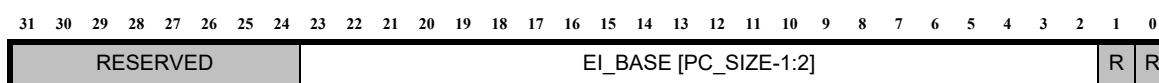


Figure 5-37 EI_BASE Register when PC_SIZE < 32



The EI_BASE register contains a base address in program memory, at which the table of instructions used by EI_S resides.

The width of the EI_BASE register is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

5.3.32 Exception Return Address, ERET

Address: 0x400

Access: RW from the mode it was last set in.

Reset 0x0000_0000

Figure 5-38 ERET Register when PC_SIZE == 32

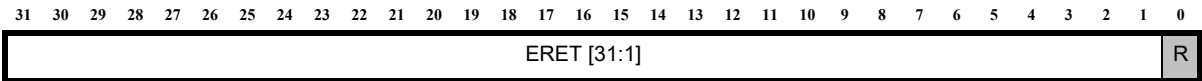
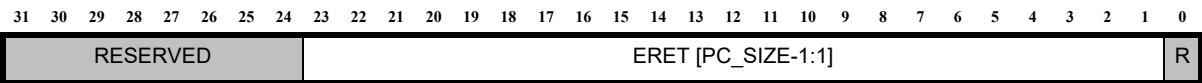


Figure 5-39 ERET Register when PC_SIZE < 32



When returning from an exception, the program counter (see [Program Counter, PC](#)) is loaded from the Exception Return Address (ERET) register. ERET is assigned a new value whenever an exception is taken. If the exception is coerced using a [TRAP_S](#) instruction, the exception return register (ERET) is loaded with the address of the next instruction to be fetched after the TRAP instruction. This value is the architectural PC expected after the TRAP completes, so any pending branches and loops are taken into account.

For a list of ERET values for each exception types, see [Table 6-7](#).

In the ARCV2-based processor, Bit 0 is ignored and must always be set to zero. When read, Bit 0 returns zero.

The width of the ERET register is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV_PrivilegeV violation exception. When an exception is triggered by an instruction in the normal mode, the internal bit is cleared to 0.

5.3.33 Exception Return Branch Target Address, ERBTA

Address: 0x401

Access: RW from the mode it was last set in.

Reset 0x0000_0000

Figure 5-40 ERBTA Register when PC_SIZE == 32

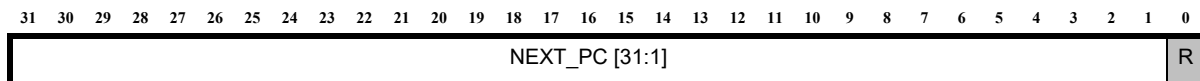
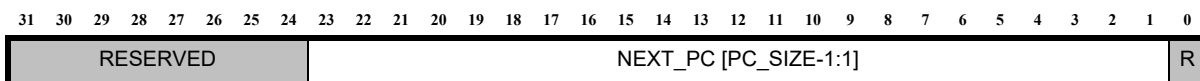


Figure 5-41 ERBTA Register when PC_SIZE < 32



When returning from an exception, the Branch Target Address register (see [Branch Target Address, BTA](#)) is loaded from the Exception Return Branch Target Address (ERBTA) register.

The width of the ERBTA register is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV_PrivilegeV violation exception. When an exception is triggered by an instruction in the normal mode, the internal bit is cleared to 0.

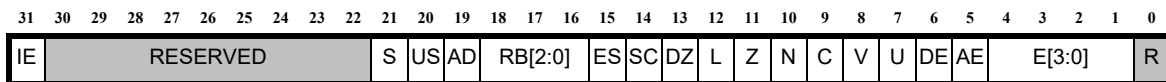
5.3.34 Exception Return Status, ERSTATUS

Address: 0x402

Access: RW from the mode it was last set in.

Reset 0x0000_0000

Figure 5-42 ERSTATUS Register



An exception saves the current status register STATUS32 register (see [Status Register, STATUS32](#)) in auxiliary register ERSTATUS. On exception entry, the STATUS32 register is copied to ERSTATUS. However, an EV_Trap exception clears the ES and DE fields before copying it to ERSTATUS, as EV_Trap is a post-commit exception.

When the [RTIE](#) instruction is executed to return from the exception handler, the current status register STATUS32 is restored from the auxiliary register ERSTATUS.

Bit 0 is ignored and must always be set to zero. When read, Bit 0 always returns zero.

When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV_PrivilegeV violation exception. When an exception is triggered by an instruction in the normal mode, the internal bit is cleared to 0.

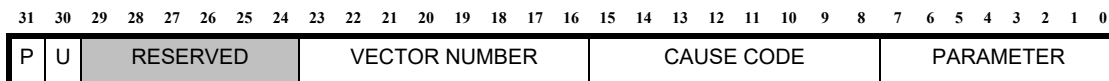
5.3.35 Exception Cause Register, ECR

Address: 0x403

Access: RW from the mode it was last set in.

Reset 0x0000_0000

Figure 5-43 ECR Register



The Exception Cause register (ECR) provides information about the source of the most recent exception to have been taken. [Figure 5-43](#) on page 186 shows the value in the Exception Cause register.

The Vector Number is an eight-bit value corresponding to the vector number used by the most recent exception. See [Table 6-5](#) on page 264 for a list of exception vectors and their assigned vector numbers.

Multiple exceptions may share each vector. The eight bit Cause Code is used to identify the exact cause of an exception and to differentiate between exceptions that share the same vector number. The actual set of exceptions raised by the processor is implementation dependent, and is therefore documented separately for each type of ARCV2 processor in [Appendix B, “Implementation-dependent Behavior”](#).

The eight bit parameter is used to pass additional information about an exception that cannot be contained in the previous fields. This is also documented separately for each type of ARCV2 processor in [Appendix B, “Implementation-dependent Behavior”](#).

ECR[29:24] bits are reserved. Reading any reserved bits returns 0 and writes to such bits have no effect.

P bit indicates that an exception occurred in an interrupt prolog.

U bit allows software to determine the User versus Kernel mode state of the processor when the most recent exception occurred.



Note

The value assigned to the U bit when an exception is taken, is implementation dependent.

See either [Appendix B.1.1, “ECR User-mode Setting in the ARC EM Family of Cores”](#) or [Appendix B.3.1, “ECR User-mode Setting for the ARC EM/EV/VPX5 Family of Cores”](#) for details of how the U bit is set for the relevant type of ARCV2 processor.

Writing to the Exception Cause register overwrites any value that has been set by the exception system. Interrupts do not set the exception cause register. Receipt of interrupts sets the appropriate ICAUSE register to the number of the last taken interrupt.

When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV_PrivilegeV

violation exception. When an exception is triggered by an instruction in the normal mode, the internal bit is cleared to 0.

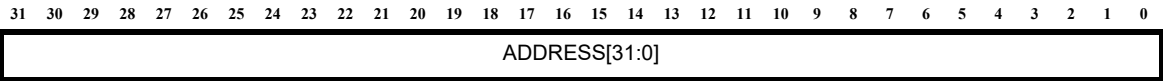
Table 5-29 ECR Field Description

Field	Bit	Reset	Description
PARAMETER	[7:0]	0	The eight bit parameter is used to pass additional information about an exception that cannot be contained in the previous fields.
CAUSE CODE	[15:8]	0	The eight bit Cause Code is used to identify the exact cause of an exception and to differentiate between exceptions that share the same vector number.
VECTOR NUMBER	[23:16]	0	The Vector Number is an eight-bit value corresponding to the vector number used by the most recent exception.
RESERVED	[29:24]	0	Reserved. Read-as-zero; ignored on write.
U	[30]	0	U bit allows software to determine the User versus Kernel mode state of the processor when the most recent exception occurred.
P	[31]	0	P bit indicates that an exception occurred in an interrupt prologue.

5.3.36 Exception Fault Address, EFA

Address:	0x404
Access:	RW from the mode it was last set in, that is, EFA set in secure mode cannot be read in non-secure mode and EFA set in non-secure mode cannot be read in secure mode.
Reset	0x0000_0000

Figure 5-44 EFA Register



This register is a baseline register, it always exists in the core.

Whenever an exception is raised, the EFA register generally records a fault address based on the exception type. For information about what value the EFA register holds for each exception, see [Table 6-7](#) on page 275.

The address captured in the EFA register, during an exception or by an auxiliary write, is limited in width to ADDR_SIZE. When ADDR_SIZE is less than 32 bits, the unused upper bits of the EFA register are read as zero and ignore on write. When the physical address extension (PAE) is enabled, the upper address bits are held in the EFA_EXT register.

□

When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV_Privilege violation exception. When an exception is triggered by an instruction in the normal mode, the internal bit is cleared to 0.

Table 5-30 EFA Field Description

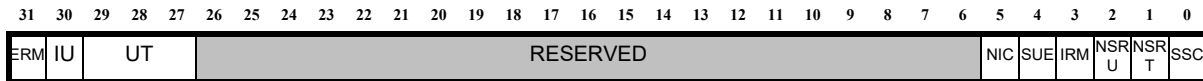
Field	Bit	Reset	Description
ADDRESS[31:0]	[31:0]	0	Depending on exception raised can hold address of data access, PC address, data-access address, physical address of the cache line being copied back, or physical address of the cache line.

5.3.37 Exception Secure Status Register, ERSEC_STAT

Address: 0x406

Access: Bit-specific; see the bit description

Figure 5-45 ERSEC_STAT



You can use this register to program the operating mode that the core must return on exception exit (from both the secure and normal modes). On exception entry, this register also saves the status of the SEC_STATUS register. On exception exit, the SEC_STAT register is restored from the ERSEC_STAT[5:0] bits.

This register exists in the build only if `-sec_modes_option==true`.

Table 5-31 ERSEC_STAT Bit-Field Definitions and Read / Write Accessibility

Field	Bit	User mode		Kernel mode		Secure User mode		Secure Kernel mode		Debug access	
		Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
SSC	0	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	Yes	Yes	Yes
NSRT	1	RAZ	IOW	RAZ	IOW	Yes	Yes	Yes	Yes	Yes	Yes
NSRU	2	RAZ	IOW	RAZ	IOW	Yes	Yes	Yes	Yes	Yes	Yes
IRM	3	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	Yes	Yes	Yes
SUE	4	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	Yes	Yes	Yes
NIC	5	RAZ	IOW	Yes	IOW	RAZ	IOW	Yes	Yes	Yes	Yes
UT micro-op type	[29: 27]	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	IOW	Yes	IOW
IU In micro-op sequence	[30]	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	IOW	Yes	IOW
ERM	31	RAZ	IOW	RAZ	IOW	RAZ	IOW	Yes	Yes	Yes	Yes

Table 5-32 ERSEC_STAT Field Description

Field	Bits	Description
SSC	[0]	Copy of the SEC_STAT register bit on exception entry.
NSRT	[1]	Copy of the SEC_STAT register bit on exception entry.
NSRU	[2]	Copy of the SEC_STAT register bit on exception entry.
IRM	[3]	Copy of the SEC_STAT register bit on exception entry.
SUE	[4]	Copy of the SEC_STAT register bit on exception entry. Writes of 1 are ignored.
NIC	[5]	Copy of the SEC_STAT register bit on exception entry.
UT	[29:27]	Indicates the micro-op sequence in progress when the exception is taken: 0x0: secure to normal call 0x1: normal to secure return 0x2: normal to secure call 0x3: secure to normal return 0x4: exception prolog 0x5: exception epilog 0x6: IRQ prolog 0x7: IRQ epilog
IU	[30]	Indicates that an exception is triggered during a micro-op sequence. 0x0: no exception is triggered during a micro-op sequence. 0x1: exception is triggered during a micro-op sequence.
ERM	[31]	Specifies the operating mode that the core must return to from an exception exit.

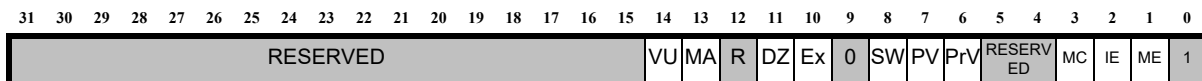
5.3.38 Secure Exception Register, AUX_SEC_EXCEPT

Address: 0x407

Access: RW in the secure mode

Default 0x00000000

Figure 5-46 AUX_SEC_EXCEPT



Use this register to program the mode (secure or normal mode) in which the exceptions are handled. The Reset vector is always handled in the secure kernel mode. The register defaults to 0: all relevant exceptions are handled in the mode they are triggered from and the core always handles the reset in the secure mode from the secure vector table reset vector. That is, if the instruction that triggered the exception is executing from a secure mode, the exception vector is handled in the secure mode, and if the instruction that triggered is executing from a normal mode, the exception vector is handled from a normal mode. Therefore, you must write two handlers for each exception: one for the normal mode and one for the secure mode.

This register exists in the build only if `-sec_modes_option==true`.

Table 5-33 AUX_SEC_EXCEPT Field Description

Field	Bits	Description
Reset	[0]	This bit is always set to 1. The Reset vector is always handled in the secure kernel mode only. You cannot program this bit. Attempts to access this bit in any other mode that secure kernel mode reads as zero and ignored on writes.
Memory Error (ME)	[1]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only.
Instruction Error (IE)	[2]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only
EV_MachineCheck (MC)	[3]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only
EV_TLBMissI	[4]	This bit is reserved. It is always read as zero and ignored on writes.
EV_TLBMissD	[5]	This bit is reserved. It is always read as zero and ignored on writes.

Table 5-33 AUX_SEC_EXCEPT Field Description (Continued)

Field	Bits	Description
EV_ProtV (PrV)	[6]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only
EV_PrivilegeV (PV)	[7]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only
EV_SWI (SW)	[8]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only
EV_Trap	[9]	This bit is always set to 0; that is, The EV_Trap exception vector is handled in the mode that triggered the exception. You cannot program this bit. Writes are ignored.
EV_Extension (EX)	[10]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only
EV_DivZero (DZ)	[11]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only
EV_DCError	[12]	This bit is reserved. It is always read as zero and ignored on writes.
EV_Misaligned (MA)	[13]	0x0: This exception vector is handled in the mode that triggered the exception. 0x1: This exception vector is handled in the secure mode only
EV_VectorUnit	[14]	This bit is reserved. It is always read as zero and ignored on writes.
Reserved	[15]	This bit is reserved. It is always read as zero and ignored on writes.

5.3.39 Interrupt Cause Registers, ICAUSE

Address: 0x40A

Access: R

Reset 0x0000_0000

Figure 5-47 ICAUSE Register



The ICAUSE register is banked and there are N copies, one corresponding to each priority level. Each banked ICAUSE register records the number of an interrupt that is taken at the register's corresponding priority level. When the AUX_IRQ_ACT register is non-zero, reading the ICAUSE register returns the interrupt number of the highest priority active interrupt (the lowest bit set in the AUX_IRQ_ACT register). When AUX_IRQ_ACT is zero (no active interrupts), the ICAUSE register read value is undefined.

If a normal interrupt is yet to be taken, reading this register returns 0×00000001 . If secure interrupts are taken, and you read the ICAUSE register corresponding to a particular priority level in the normal kernel mode, the last normal interrupt number that was taken at this priority level is returned. A read of an ICAUSE register returns the interrupt number irrespective of whether the interrupt is a secure interrupt or not.

This register is present in a build only if the processor is configured to include interrupts. The size of this register depends on the number of interrupts configured in the system. If M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$, where ceil is the function to round to the next higher integer.

On reset, this register contains 0×00000001 for configured interrupts; if Interrupt [m-1:0] is greater than M+16-1 or less than 16, this register is read as zero. Where, M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$, and ceil is the function to round to the next higher integer.

Table 5-34 ICAUSE Field Description

Field	Bit	Reset	Description
ICAUSE[m-1:0]	[m-1:0]	0	Triggered interrupt number
RESERVED	[31:m]	0	Reserved. Read-as-zero; ignored on write.

5.3.40 Interrupt Select, IRQ_SELECT

Address: 0x40B

Access: RW

Reset 0x0000_0000

Figure 5-48 IRQ_SELECT Register



This register allows software to select a bank of registers, to read or write, that are associated with a specific interrupt number. Interrupts occupy vector numbers 16 up to 255 depending on the configuration. Vector numbers 0 – 15 are used for exceptions. You can read and write to the IRQ_SELECT register in kernel mode; this register is protected and inaccessible in user mode.

The width of the Interrupt field in this register is determined by the number of interrupts configured in the system. If M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$, where ceil is the function to round to the next higher integer. This register is present in a build only if the processor is configured to include interrupts. On reset, this register contains 0x00000000.

Table 5-35 IRQ_SELECT Field Description

Field	Bit	Reset	Description
INTERRUPT[m-1:0]	[m-1:0] If M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$	0	Selects a specific interrupt. If Interrupt [m-1:0] is greater than $M+16-1$ or less than 16, all banked IRQ auxiliary registers are read as zero and ignored on write.
RESERVED	[31:m]	0	Reserved. Read-as-zero; ignored on write.

5.3.41 Interrupt Enable Register, IRQ_ENABLE

Address: 0x40C

Access: RW

Reset 0x0000_0000

Figure 5-49 IRQ_ENABLE Register

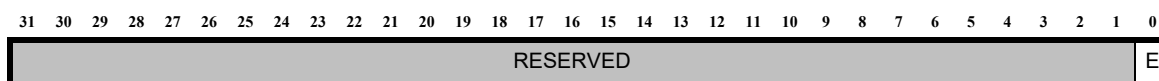


Table 5-36 IRQ_ENABLE Field Description

Field	Bit	Reset	Description
E	[0]	0	Value of IRQ_ENABLE for the selected interrupt.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

This banked register allows software to enable or disable an interrupt selected by the [Interrupt Select, IRQ_SELECT](#) register.

Secure interrupts can be enabled only from the secure kernel mode. Attempts to enable secure interrupts from the normal kernel mode are ignored (read as zero and ignored on writes). You can enable normal interrupts from the normal and secure kernel modes.

If an interrupt enable of 0 is assigned, the associated interrupt is disabled. If an interrupt enable of 1 is assigned, the associated interrupt is enabled. This register is present in a build only if the processor is configured to include interrupts.

On reset, this register contains 0x00000001 for configured interrupts. Therefore, all configured interrupts are individually enabled upon reset. If Interrupt [m-1:0] is greater than M+16-1 or less than 16, this register is read as zero and ignored on write. Where, M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$, and ceil is the function to round to the next higher integer.

5.3.42 Interrupt Trigger Register, IRQ_TRIGGER

Address: 0x40D

Access: ■ When SecureShield 2+2 mode is configured: RW

- RW in the normal and secure modes for normal interrupts
- RW in the secure mode for the secure interrupts

Reset 0x0000_0000

Figure 5-50 IRQ_TRIGGER Register

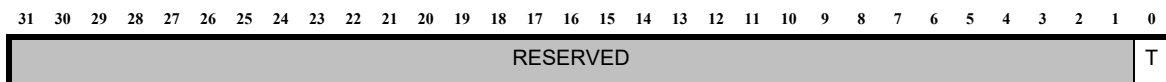


Table 5-37 IRQ_TRIGGER Field Description

Field	Bit	Reset	Description
T	[0]	0	Value of IRQ_TRIGGER, for the selected interrupt
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

This banked register allows software to set and examine the current level or pulse trigger setting for the IRQ selected by the [Interrupt Select, IRQ_SELECT](#) register.

For secure interrupts, the corresponding IRQ_TRIGGER register can be accessed from the secure kernel mode only. For secure interrupts, attempts to access the corresponding IRQ_TRIGGER register from the normal kernel mode are ignored (read as zero and ignored on writes). However, in the secure kernel mode, you can access the IRQ_TRIGGER register of both secure and normal interrupts.

If an interrupt trigger value of 0 is assigned, the associated interrupt is level sensitive. If an interrupt trigger of 1 is assigned, the associated interrupt is pending up to one cycle after a transition from 0 to 1 is detected on the interrupt line that is sampled at CPU clock frequency. The interrupt remains pending until you write a 1 to the corresponding [Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL](#) register.

This register is present in a build only if the processor is configured to include interrupts. When interrupts are included, this register only exists for external interrupts.

On reset, this register contains 0x00000000 for configured interrupts; if Interrupt [m-1:0] is greater than M+16-1 or less than 16, this register is read as zero and ignored on write. Where, M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$, and ceil is the function to round to the next higher integer.

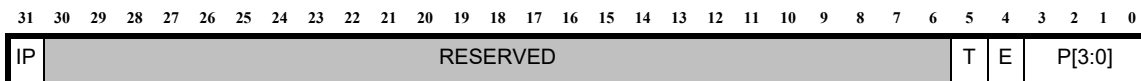
5.3.43 Interrupt Status Register, IRQ_STATUS

Address: 0x40F

Access: R

Reset 0x0000_0000

Figure 5-51 IRQ_STATUS Register



This banked register allows software to examine all current status information for the IRQ selected by the [Interrupt Select, IRQ_SELECT](#) register.

For secure interrupts, the corresponding IRQ_STATUS register can be read from the secure kernel mode only. For secure interrupts, attempts to read the corresponding IRQ_STATUS register from the normal kernel mode are ignored (read as zero). However, in the secure kernel mode, you can read the IRQ_STATUS register of both secure and normal interrupts.

The reset value is determined by the reset value of the associated individual interrupt registers. This register is present in a build only if the processor is configured to include interrupts.

Table 5-38 IRQ_STATUS Field Description

Field	Bit	Reset	Description
P[3:0]	[3:0]	0	Value of Interrupt Priority Register, IRQ_PRIORITY , for the selected interrupt
E	[4]	0	Value of Interrupt Enable Register, IRQ_ENABLE , for the selected interrupt
T	[5]	0	Value of Interrupt Trigger Register, IRQ_TRIGGER , for the selected interrupt
RESERVED	[30:6]	0	Reserved. Read-as-zero; ignored on write.
IP	[31]	0	Value of Interrupt Pending Register, IRQ_PENDING , for the selected interrupt. Bit 31 allows quick check for pending interrupt using the .MI condition or less-than-zero test.

5.3.44 User Mode Extension Enable Register, XPU

Address: 0x410

Access: RW

Figure 5-52 XPU Register

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
u31	u30	u29	u28	u27	u26	u25	u24	u23	u22	u21	u20	u19	u18	u17	u16	u15	u14	u13	u12	u11	u10	u9	u8	u7	u6	u5	u4	u3	u2	u1	u0

The User Mode Extension Enable register (XPU) controls User-mode access to extension instructions and state. Each extension group is assigned a bit within the XPU register, and this bit may be programmed to enable or disable User-mode access to the extensions within that group. If the bit is set to 1, all extension features in the corresponding extension group are accessible in User mode. If the bit is set to 0, those features are inaccessible in User mode. A disabled extension exception is raised if an attempt is made to access any extension feature in User mode, if its corresponding XPU bit is set to 0.

When `-sec_modes_option==true`, you can configure extensions to be executed in the secure mode only. For secure-mode only extensions, the corresponding XPU bits are read as zero and ignored on writes from the normal mode.



Note

You cannot configure extensions with different modes (secure or normal) to use the same XPU bit; otherwise this XPU bit will be RAZ/WI is read as zero and ignored on writes when accessed in the normal kernel mode.

This register allows the following capabilities to be supported in software:

- Disabling of extension functions. For example, to permit software emulation of extensions to be tested
- Operating systems to grant User-mode access to extension functions and state
- Intelligent context switching of extension state (lazy context switch)
- Context switching of extension hardware in system containing re-configurable logic

When a Privilege Violation exception is raised due to a User-mode attempt to access a protected extension, the XPU bit number of the faulting exception group is assigned to the parameter field of the exception cause register (see Exception Cause Register, ECR). This allows an OS to do the following:

- Distinguish between an access to a protected extension and a non-existent extension.
- For a protected extension, determine which extension group was accessed.

Four categories of extension feature are protected by this mechanism. They are: extension instructions, extension condition codes, extension core registers, and extension auxiliary registers. If Privilege Violations are raised due to references to several different extension groups, all by the same instruction, the ECR parameter field is set to the extension group number of the highest priority violating feature. In this context, the priority ordering is:

1. Instruction extensions
2. Extension condition codes
3. An extension core register in a non-load destination context
4. An extension core register used to return a value loaded from memory
5. An extension core register used as the first source register (the B operand register)
6. An extension core register used as the second source register (the C operand register)
7. An extension auxiliary register access

On Reset, the XPU register is set to 0x00000000, so that all extension groups are initially protected from User mode access. For more information about the allocation of XPU bits to each extension group, see the processor-specific Databook.

5.3.45 Branch Target Address, BTA

Address: 0x412

Access: RW

Reset 0x0000_0000

Figure 5-53 BTA Register (PC_SIZE == 32)

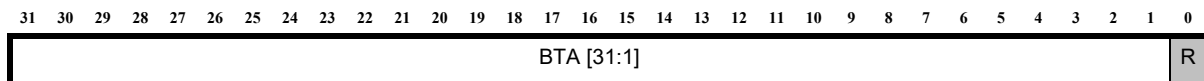
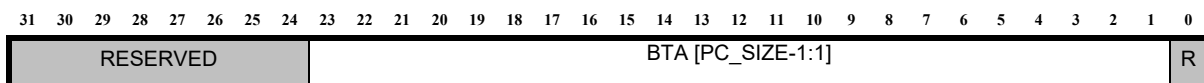


Figure 5-54 BTA Register (PC_SIZE < 32)



The BTA register is updated in the following situations:

- A branch or jump with a delay slot is taken
- Returns from exceptions that occur between a branch or jump and any associated delay slot instruction
- When an EI_S (Execute Indexed) instruction is executed
- When written by an SR instruction

Branch or Jump Instruction with a Delay Slot

When a taken branch or jump with a delay slot is committed, the BTA register is loaded with the target address of the branch, and status bit STATUS32.DE bit is set signifying that a delayed branch is pending. The BTA register holds the program counter value to be used after the next delay slot instruction has committed. This information allows an exception to be taken between the branch or jump and its delay slot instruction. When this happens, the BTA register is saved in ERBTA and the STATUS32 register, including STATUS32.DE, is saved in ERSTATUS. On return from the exception, when the processor restores these registers, if the restored STATUS32.DE bit is set, it indicates to the processor that a branch has committed and is pending execution of the delay-slot instruction. Thus, after the delay-slot instruction commits, the program counter is loaded from the BTA register and execution resumes from the branch target address.

Note that STATUS32.DE is set only if the branch is taken, and is cleared if the branch is not taken or if the instruction is not a taken branch or jump.

Returning from Exceptions Between a Branch or Jump

Exceptions are permitted between a branch or jump and any associated delay slot instruction. Therefore, a special branch target address register, ERBTA is provided to retain a copy of the BTA register when such exceptions occur.

When returning from an exception, if the STATUS32[DE] bit (see [Status Register, STATUS32](#)) is set to 1 as a result of the RTIE operation, the value in the BTA register is restored from the Exception Return BTA register (see [Exception Return Branch Target Address, ERBTA](#)), allowing the program to resume execution at the correct point. The RTIE instruction is somewhat special, in that it restores STATUS32.DE to a previous value before returning from the interrupt or exception context in which it is executed. The rules are described in [Status Register, STATUS32](#).

When an EI_S instruction is Executed

The Execute Indexed instruction, EI_S, causes the next instruction to be fetched from a table of instructions, located in memory at an address given by the EI_BASE auxiliary register. The instruction fetched from the table is said to be in the execution slot of the EI_S instruction. Not all instructions may be executed in an execution slot; for example another EI_S instruction, or any type of branch or jump instruction, are all illegal in that context. When an EI_S instruction is executed it sets the STATUS32.ES bit, to indicate that the next instruction is evaluating in an execution slot. Therefore, when STATUS32.ES is set to 1, the rules restricting the set of instructions that can be executed in an execution slot are applied. It may not be that the previous instruction was an EI_S, as for example an exception may have occurred the first time the execution-slot instruction was attempted. In this case, the STATUS32.ES bit is restored to 1 when returning from the exception, and the execution slot instruction may be attempted a second time. When the STATUS32.ES bit is set to 1, the next PC value is always obtained from the BTA auxiliary register. This register is set to point at the next instruction after the EI_S when it executes, thereby ensuring that the execution-slot instruction is always followed by the next instruction after the EI_S.

5.3.46 Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL

Address: 0x415

Access: W

Reset 0x0000_0000

Figure 5-55 IRQ_PULSE_CANCEL Register

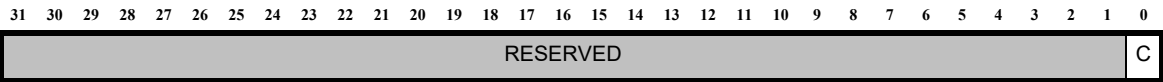


Table 5-39 IRQ_PULSE_CANCEL Field Description

Field	Bit	Reset	Description
C	[0]	0	Value to be written to IRQ_PULSE_CANCEL for the selected interrupt.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

Pulse-triggered interrupts are generally auto clearing. If you choose not to take a pulse-triggered interrupt, you can use this banked register to clear that pulse-triggered interrupt for the IRQ selected by the [Interrupt Select, IRQ_SELECT](#) register.

Reset value is not applicable, as this register cannot be read. A write of pulse-canceling value 0 is always ignored. A write of pulse-canceling value 1 to an non-level sensitive interrupt clears any saved interrupt.

This register is present in a build only if the processor is configured to include interrupts. When interrupts are included, this register only exists for external interrupts.

For secure interrupts, the corresponding IRQ_PULSE_CANCEL register can be written from the secure kernel mode only. For secure interrupts, attempts to write the corresponding IRQ_PULSE_CANCEL register from the normal kernel mode are ignored. However, in the secure kernel mode, you can write to the IRQ_PULSE_CANCEL register of both secure and normal interrupts.

5.3.47 Interrupt Pending Register, IRQ_PENDING

Address: 0x416

Access: R

Reset 0x0000_0000

Figure 5-56 IRQ_PENDING Register

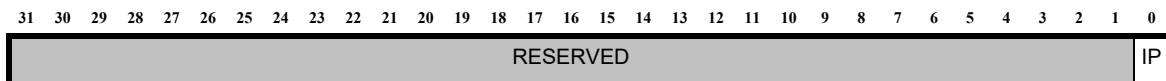


Table 5-40 IRQ_PENDING Field Description

Field	Bit	Reset	Description
IP	[0]	0	Value of IRQ_PENDING for the selected interrupt
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

This banked register allows software to detect if there is a pending interrupt for the IRQ selected by the [Interrupt Select, IRQ_SELECT](#) register. This register includes pending interrupts asserted by the AUX_IRQ_HINT register. This register is present in a build only if the processor is configured to include interrupts.

The IRQ pending bit is asserted when a pulse sensitive IRQ is detected or a level-sensitive IRQ signal is asserted regardless of the global enable bit in STATUS32 auxiliary register (STATUS32.IE) or individual IRQ enable setup.

For pulse sensitive IRQs, this pending bit can be cleared by software by writing to IRQ_PULSE_CANCEL aux register; the pending bit is also auto cleared when this IRQ is handled, that is, when PC enters the IRQ handler for this pulse sensitive IRQ.

For level sensitive IRQs, the pending bit follows the IRQ source, that is, when IRQ signal source is de-asserted, this pending bit is also de-asserted; the pending bit for level-sensitive IRQs strictly follows the external IRQ signal; this pending bit cannot be cleared by writing to the ARC auxiliary registers.

For secure interrupts, the corresponding IRQ_PENDING register can be read from the secure kernel mode only. For secure interrupts, attempts to read the corresponding IRQ_PENDING register from the normal kernel mode are ignored (read as zero). However, in the secure kernel mode, you can read the IRQ_PENDING register of both secure and normal interrupts.

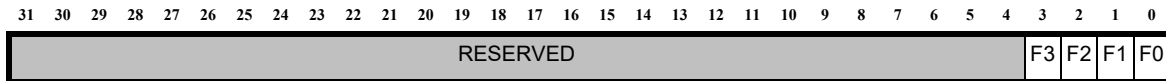
On reset, this register contains 0x00000000 for configured interrupts; if Interrupt [m-1:0] is greater than M+16-1 or less than 16, this register is read as zero. Where, M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$, and ceil is the function to round to the next higher integer.

5.3.48 User Extension Flags Register, XFLAGS

Address: 0x44F

Access: RW

Figure 5-57 User Extension Flags Register, XFLAGS



The User Extension Flags register (XFLAGS) is present in a build of an ARCV2-based processor when at least one APEX component is included. The four extension flags are available for any extension instruction to use as an implicit input, and to define as an implicit output. Only the lower 4 bits of this register are implemented. Bits 4 to 31 are reserved. On reset, the XFLAGS register is set to 0.

The XFLAGS register can be read or written in Kernel mode, but is protected from both read and write in User mode.

5.4 Build Configuration Registers

The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

A set of baseline Build Configuration Registers are present in all ARCV2 processors. In addition, each optional ISA extension typically includes a Build Configuration Register to signify its presence and its specific configuration in each given build.

Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCV2-based system.

Auxiliary registers, in the range 0x60 to 0x7F and 0xC0 to 0xFF, are assumed to be BCRs. In kernel mode, any read from a non-existent build configuration register in this range returns 0, and no exception is generated. This design enables the kernel-mode code to detect the presence or absence of a BCR because all BCRs that are present in a system contain non-zero values.

However, in user mode, reads from build configuration registers always raise a Privilege Violation exception (see [Privilege Violation, Kernel/User Only Access](#)).

Any write to a build configuration register, whether in kernel or user mode, raise an [Illegal Instruction](#) exception.

[Table 5-41](#) summarizes the build configuration registers for components that are described in this document.

Table 5-41 Build Configuration Registers

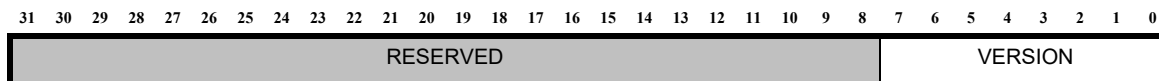
Number	Name	Access
0x60	Build Configuration Registers Version, BCR_VER	R
0x63	BTA Configuration Register, BTA_LINK_BUILD	R
0x68	Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD	R
0x70	Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD	SR
0x6E	Core Register File Configuration Register, RF_BUILD	R
0x7B	Multiplier Configuration Register, MULTIPLY_BUILD	R
0x7C	Swap Instruction Configuration Register, SWAP_BUILD	R
0x7D	Normalize Instruction Configuration Register, NORM_BUILD	R
0x7E	Min/Max Instruction Configuration Register, MINMAX_BUILD	R
0x7F	Barrel Shifter Configuration Register, BARREL_BUILD	R
0xC1	Instruction Set Configuration Register, ISA_CONFIG	R
0xF3	Interrupt Build Configuration Register, IRQ_BUILD	R

5.4.1 Build Configuration Registers Version, BCR_VER

Address: 0x60

Access: R

Figure 5-58 BCR_VER Register



The Build Configuration Registers Version (BCR_VER) specifies which build configuration register implementation is present.

Two regions in the auxiliary address space are dedicated to BCRs at 0x60-0x7F and 0xC0-0xFF. The BCR_VER register indicates which, if any, of these two regions are implemented by each specific configuration of an ARCV2-based processor.

Table 5-42 BCR_VER Field Descriptions

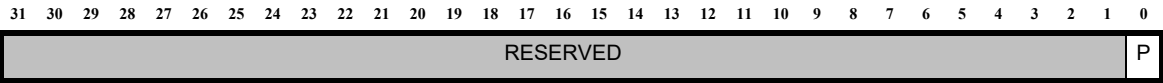
Field	Bit	Description
VERSION	[7:0]	Version of Build Configuration Registers. All non-enumerated values are reserved. ■ 0x1: Indicates that the BCR Region is at addresses 0x60-0x7F and ISA_CONFIG only ■ 0x2: Indicates that the BCR Region is at addresses 0x60-0x7F and 0xC0-0xFF ■ 0x3: Indicates that the BCR Region is at address 0x60-0x7F, 0xC0-0xFF, and 0xF60 and 0xFFF
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

5.4.2 BTA Configuration Register, BTA_LINK_BUILD

Address: 0x63

Access: R

Figure 5-59 BTA_LINK_BUILD Register



The BTA configuration register, BTA_LINK_BUILD, specifies the set of Branch Target Address (BTA) registers that are present in the architecture.

Table 5-43 BTA_LINK_BUILD Field Descriptions

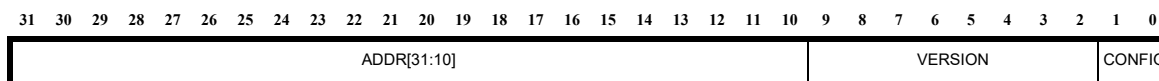
Field	Bit	Description
P	[0]	Presence of BTA Registers. All non-enumerated values are reserved. 0x0 = BTA_L1 and BTA_L2 registers are absent
RESERVED	[31:1]	Reserved. Read-as-zero; ignored on write.

5.4.3 Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD

Address: 0x68

Access: R

Figure 5-60 VECBASE_AC_BUILD Register



The default base address of the interrupt vector table is fixed when a particular ARCV2-based system is created. On [Reset](#), the programmable vector base register (see [Interrupt Vector Base Register, INT_VECTOR_BASE](#)) is set from the constant value in VECBASE_AC_BUILD.

Bits [31:10] show the interrupt vector base address based on the configuration of the interrupt system.

Table 5-44 VECBASE_AC_BUILD Field Descriptions

Field	Bit	Description
CONFIG	[1:0]	Configuration of the reset value for INT_VECTOR_BASE. All non-enumerated values are reserved. ■ 0x0: Address is supplied with the <code>-intvbase_preset</code> option, when <code>-intvbase_ext</code> is false. ■ 0x1: Address is supplied through the <code>intvbase_in</code> core input signal, when the <code>-intvbase_ext</code> option is true. ■ . Other values are reserved.
VERSION	[9:2]	Version of Interrupt Unit. All non-enumerated values are reserved. ■ 0x04 = ARCV2 Interrupt Unit
ADDR[31:10]	[31:10]	Interrupt vector base address, excluding the lower 10 bits, that is configured when the core comes out of reset.

Table 5-45 Value of the CONFIG[1:0] Field based on Configuration Options

<code>-intvbase_ext</code>	<code>-sec_modes</code>	CONFIG[1:0]
True	False	0x1: Address (VECBASE_AC_BUILD[ADDR]) is supplied through the <code>intvbase_in</code> core input signal.
X	True	0x0: Address (VECBASE_AC_BUILD[ADDR]) is supplied with the <code>-intvbase_preset</code> option.

5.4.4 Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD

Address: 0x70

Access: R in the secure mode

Figure 5-61 SEC_VECBASE_BUILD Register

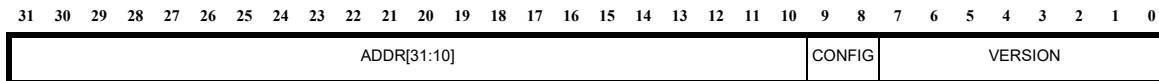


Table 5-46 SEC_VECBASE_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of Interrupt Unit with secure interrupt table 0x01 = ARCV2 Interrupt Unit
CONFIG	[9:8]	Configuration of the reset value for INT_VECTOR_BASE_S: Only values of 0x0 and 0x1 are supported. See Table 5-47 for the description. Other values are reserved.
ADDR	[31:10]	Base Address of the secure interrupt table. The value of the ADDR field is configured at build time through the - intvbase_preset_s option.

Table 5-47 Value of the CONFIG[9:8] Field based on Configuration Options

-intvbase_ext	-sec_modes	CONFIG[1:0]
True	True	0x1: Address (SEC_VECBASE_BUILD[ADDR]) field is supplied through the intvbase_in core input signal.
False	True	0x0: Address (SEC_VECBASE_BUILD[ADDR]) is supplied by the -intvbase_preset_s option.
X	False	This SEC_VECBASE_BUILD register does not exist.

This register exists in the build only when `-sec_modes_option==true`.

The default base address of the secure interrupt vector table is fixed when a particular ARCV2-based system is created. On [Reset](#), the programmable vector base register (see [Secure Interrupt Vector Base Register, INT_VECTOR_BASE_S](#)) is set from the constant value in SEC_VECBASE_BUILD.

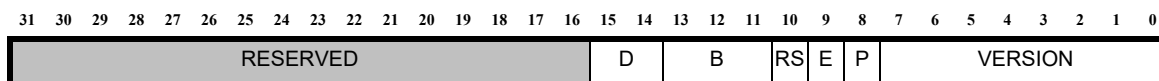
Bits 31 to 10 show the interrupt vector base address based on the configuration of the interrupt system.

5.4.5 Core Register File Configuration Register, RF_BUILD

Address: 0x6E

Access: R

Figure 5-62 RF_BUILD Register



The Core Register File Configuration register (RF_BUILD) determines whether the base core register set is configured as a 16 or 32 entry set, and whether the register set is cleared on Reset. The RF_BUILD register also indicates whether the register set is made up from a 3-port or 4-port register file, and if any additional register banks are available.

Table 5-48 RF_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of Core Register Set 0x2: Current Version
P	[8]	Number of Ports 0x0: 3-port register file 0x1: 4-port register file
E	[9]	Number of Entries 0x0: 32-entry register file 0x1: 16-entry register file
RS	[10]	Reset State 0x0: Not cleared on reset 0x1: Cleared on reset
B	[13:11]	Number of Register Banks in addition to the main core register bank 0x0: 0 additional register bank 0x1: 1 additional register banks 0x2: 2 additional register bank 0x3: 3 additional register banks 0x4: 4 additional register bank 0x5: 5 additional register banks 0x6: 6additional register bank 0x7: 7 additional register banks

Table 5-48 RF_BUILD Field Descriptions (Continued)

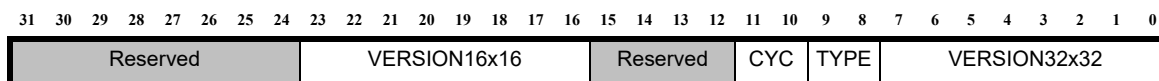
Field	Bit	Description
D	[15:14]	Number of duplicated registers in each additional register bank ■ 0x0: 4 duplicated registers ■ 0x1: 8 duplicated registers ■ 0x2: 16 duplicated registers ■ 0x3: 32 duplicated registers Note: This field returns 0 when RF_BUILD.B == 0, that is there are no additional register banks.
RESERVED	[31:16]	Reserved. Read-as-zero; ignored on write.

5.4.6 Multiplier Configuration Register, MULTIPLY_BUILD

Address: 0x7B

Access: R

Figure 5-63 MULTIPLY_BUILD Register



The Multiplier configuration register (MULTIPLY_BUILD) contains the version of the multiply instructions.

Table 5-49 MULTIPLY_BUILD Field Descriptions

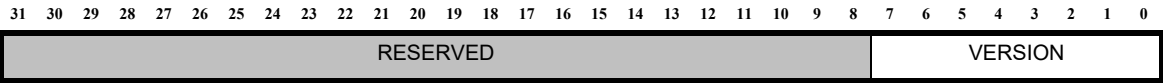
Field	Bit	Description
VERSION32x32	[7:0]	Version of 32x32 Multiply. All non-enumerated values are reserved. 0x6: ARCV2 32x32 Multiply
TYPE[1:0]	[9:8]	Type of Multiply. All non-enumerated values are reserved. 0x0: Serial multiplier, computing 4-bits of the result per cycle 0x1: Parallel multiplier, with latency defined in the CYC field. 0x2: Parallel multiplier, with DSP capabilities defined by the DSP field.
CYC[1:0]	[11:10]	Number of Cycles 0x0: 1 cycle 0x1: 2 cycles 0x2: 3 cycles 0x3: 4 cycles
RESERVED	[15:12]	Reserved. Read-as-zero; ignored on write.
DSP [3:0]	[15:12]	DSP capabilities. All non-enumerated values are reserved. 0x0: No DSP capabilities included 0x1: Supports DSP features from MPY_OPTION 7 0x2: Supports DSP features from MPY_OPTION 8 0x3: Supports DSP features from MPY_OPTION 9
VERSION16x16	[23:16]	Version of 16x16 Multiply. All non-enumerated values are reserved. 0x02: ARCV2 16x16 Multiply
RESERVED	[31:24]	Reserved. Read-as-zero; ignored on write.

5.4.7 Swap Instruction Configuration Register, SWAP_BUILD

Address: 0x7C

Access: R

Figure 5-64 SWAP_BUILD Register



The Swap instruction configuration register (SWAP_BUILD) contains the version of the [SWAP](#) option.

Table 5-50 SWAP_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of Swap. All non-enumerated values are reserved. 0x1 : ARCompact (ARCV1) version of SWAP option 0x3: ARCV2 version of SWAP option
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

5.4.8 **Normalize Instruction Configuration Register, NORM_BUILD**

Address: 0x7D

Access: R

The Normalize instruction configuration register (NORM_BUILD) contains the version of the normalize instructions [NORM](#) and NORMH (see [NORMH NORMW](#)).

Figure 5-65 NORM_BUILD Register

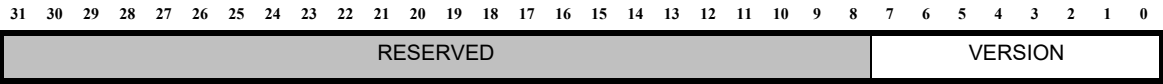


Table 5-51 NORM_BUILD Field Descriptions

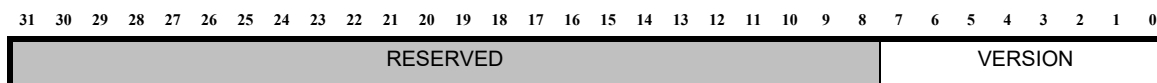
Field	Bit	Description
VERSION	[7:0]	Version of Norm. All non-enumerated values are reserved. ■ 0x1 : Reserved ■ 0x2 : ARCompact (ARCV1) version ■ 0x3 : ARCV2 version
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

5.4.9 Min/Max Instruction Configuration Register, MINMAX_BUILD

Address: 0x7E

Access: R

Figure 5-66 MINMAX_BUILD Register



The MIN/MAX instruction configuration register, MINMAX_BUILD, contains the version of the [MIN](#) and [MAX](#) instructions.

Table 5-52 MINMAX_BUILD field descriptions

Field	Bit	Description
VERSION	[7:0]	Version of Min/Max. All non-enumerated values are reserved. 0x1 : Reserved 0x2 : Current Version
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

5.4.10 Barrel Shifter Configuration Register, BARREL_BUILD

Address: 0x7F

Access: R

The Barrel Shifter Configuration register (BARREL_BUILD) contains the version of the barrel shift and rotate instructions.

Figure 5-67 BARREL_BUILD Register

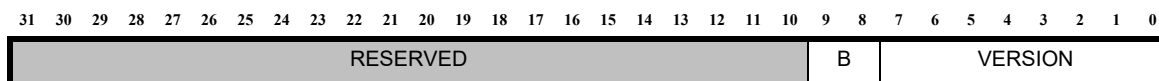


Table 5-53 BARREL_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of Barrel Shifter. All non-enumerated values are reserved. 0x1: Reserved 0x2: ARCompact (ARCV1) version 0x3: ARCV2 version
B[1:0]	[9:8]	SHIFT_OPTION 0x0 : Single-bit shifts only 0x1: Single-bit and shift-assist instructions 0x2: Single-bit and barrel-shift instructions 0x3: All shift instructions
RESERVED	[31:10]	Reserved. Read-as-zero; ignored on write.

5.4.11 Instruction Set Configuration Register, ISA_CONFIG

Address: 0xC1

Access: R

The ISA_CONFIG auxiliary register is a read-only build configuration register (BCR) that is present in every ARCV2-based processor, regardless of its configuration. This register summarizes the build configuration with respect to the programming model options and some of the ISA options defined in the section on “Configurability” on page 105.

In addition to the configured build options, the version field indicates the current version number for the ISA_CONFIG register. This design enables future additions to the ISA_CONFIG register fields.

The values of each field in ISA_CONFIG select the configuration of the relevant option in accordance with the interpretation defined by the section on “Configurability” on page 105, and Table 3-2.

Figure 5-68 ISA_CONFIG Build Register

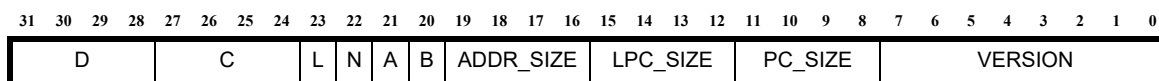


Table 5-54 ISA_CONFIG Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Indicates the current version for the ISA_CONFIG register 0x1: Version of the ISA_CONFIG register in ARCompact (ARCV1) ISA-based processors 0x2: Version of the ISA_CONFIG register in ARCV2 ISA-based processors.
PC_SIZE	[11:8]	PC size option. All non-enumerated values are reserved. 0x0: 16-bit width 0x1: 20-bit width 0x2: 24-bit width 0x3=28-bit width 0x4=32-bit width

Table 5-54 ISA_CONFIG Field Descriptions (Continued)

Field	Bit	Description
LPC_SIZE	[15:12]	Zero Overhead loop option. All non-enumerated values are reserved. 0x0: Zero overhead loop not supported 0x1: 8-bit width 0x2: 12-bit width 0x3: 16-bit width 0x4: 20-bit width 0x5: 24-bit width 0x6: 28-bit width 0x7: 32-bit width
ADDR_SIZE	[19:16]	ADDR_SIZE option. All non-enumerated values are reserved. 0x0: 16-bit width 0x1: 20-bit width 0x2: 24-bit width 0x3: 28-bit width 0x4: 32-bit width
B	[20]	BYTE_ORDER 0x0: Little-Endian byte ordering 0x1: Big-Endian byte ordering
A	[21]	ATOMIC_OPTION 0x0: LLOCK and SCND instructions are absent 0x1: LLOCK and SCND instructions are present
N	[22]	NON_ALIGNED 0x0: All data memory references must use natural alignment 0x1: Non-aligned memory references are supported
L	[23]	LL64_OPTION 0x0: LDD and STD extension instructions are not included in the build 0x1: LDD and STD instructions are included in the build
C	[27:24]	Version of the code density unit included in the processor. All non-enumerated values are reserved. 0x1: Code density unit version 1 0x2: Code density unit version 2

Table 5-54 ISA_CONFIG Field Descriptions (Continued)

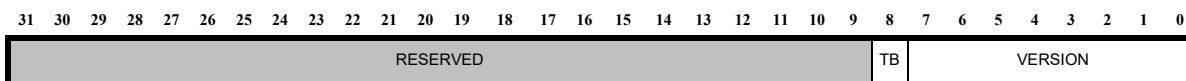
Field	Bit	Description
D (DIV_REM_OPTION)	[31:28]	DIV_REM_OPTION. All non-enumerated values are reserved. ■ 0x0: DIV/REM instructions are absent ■ 0x1: Bit-serial DIV, DIVU, REM and REMU implementation ■ 0x2: Radix-4 fast DIV, DIVU, REM and REMU implementation

5.4.12 Core Architecture Build Configuration Register, ARCH_CONFIG

Address: 0xF1

Access: R

Figure 5-69 ARCH_CONFIG Register



This register specifies the build-time configuration of the turbo boost option. When you have configured the turbo boost option, the achievable clock frequency is increased, but at the cost of an additional cycle latency on branch instructions.

Table 5-55 ARCH_CONFIG Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version 0x01: First version
TB	[8]	Specifies the build-time configuration of the turbo boost option. 0x0: Turbo boost is not configured. 0x1: Turbo boost is configured.
RESERVED	[31:9]	Reserved. Read-as-zero; ignored on write.

5.4.13 Interrupt Build Configuration Register, IRQ_BUILD

Address: 0xF3

Access: R

This register allows software to see how many interrupts are configured within the Interrupt Controller, how many of those interrupts have external interrupt pins, and whether the non-maskable imprecise exception is configured.

Figure 5-70 IRQ_BUILD Register

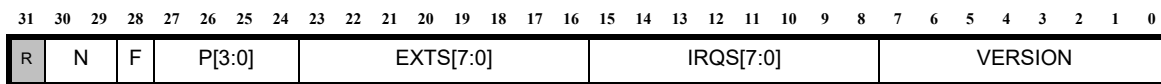


Table 5-56 IRQ_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of the Interrupt Controller 0x1 = ARC v2 0x2: Indicates support for the interrupt unit that supports the multi-core NMI feature. To determine the NMI feature configuration, see IRQ_BUILD[30:29] field.
IRQS	[15:8]	Indicates the number of interrupts configured in the Interrupt Controller.
EXTS	[23:16]	Indicates the number of external interrupt lines configured in the Interrupt Controller.
P	[27:24]	Contains N-1, when N interrupt priority levels are configured.
F	[28]	Value of the FIRQ_OPTION configuration option
N	[30:29]	Indicates whether the non-maskable imprecise exception is configured. 0x0: not configured. 0x1: single-core non-maskable imprecise exception is configured. 0x2: multi-core cluster-level non-maskable imprecise exception is configured. The following registers are added to the processor when this option is configured: - CC_NMI_STATUS - CC_NMI_CAUSE This bit reflects the value of the <code>-nmi_option</code> configuration option.
RESERVED	[31]	Reserved. Read-as-zero; ignored on write.



Note

The reset value of this register is defined by the configuration of the interrupt controller, which sets the number of interrupts and the number of those interrupts that are externally visible as interrupt lines. If the non-maskable imprecise exception is supported, but the number of configured interrupts is 0, then the IRQ_BUILD register contains the value 0x20000001.

6

Interrupts and Exceptions

6.1 Introduction

ARCV2-based processors support exceptions and may optionally be configured with interrupts. Exceptions are synchronous to an instruction. Most exceptions can occur at the same place each time a program is executed (apart from a Memory Error exception that can occur asynchronously), whereas interrupts are typically asynchronous.

Interrupts and exceptions cause the processor to enter the kernel operating mode. Upon returning from an interrupt or an exception, the processor enters the operating mode defined by the saved state of the processor at the time of interrupt or exception entry. Alternatively, the interrupt or exception handler may modify the saved state to explicitly set the operating mode on return from interrupt or exception.

In a processor configured with the `-sec_modes_option==true`, exceptions and interrupts can be handled either in the secure or normal mode. Secure interrupts and exceptions cause the processor to enter the secure kernel operating mode, and normal interrupts and exception cause the processor to enter normal kernel operating mode. The processor, upon returning from an interrupt or an exception, returns to the operating mode defined by the saved state of the processor at the time of interrupt or exception entry.

When secure modes are enabled, some restrictions are enforced based on the `ER_SECSTAT` registers. For example, a normal exception cannot force a return to a secure mode.

6.2 Privileges and Operating Modes

The operating mode of the processor determines whether a task is permitted to execute a privileged instruction or access protected state. For example, the operating mode is used by the memory management system, if present, to determine whether a specific location in memory is accessible.

Two operating modes are provided: Kernel mode and User mode.

The kernel-mode tasks use the bits in the `STATUS32` register (see [Status Register, STATUS32](#)) to determine the operating mode of the processor to correctly recover from all legitimate interrupt or exception situations, and to enable the complete processor state to be saved and restored.

Following are the operating modes:

- Secure kernel mode (available with `-sec_modes_option==true`)
- Secure user mode (available with `-sec_modes_option==true`)
- normal kernel mode

■ normal user mode

The secure kernel-mode tasks use the bits in the STATUS32, SEC_STAT, and the ER_SECSTAT registers to determine the operating mode of the processor to correctly recover from all legitimate interrupt or exception situations, and to enable the complete processor state to be saved and restored.

The normal kernel-mode tasks use the bits in the STATUS32 register to determine the operating mode of the processor to correctly recover from all legitimate interrupt or exception situations, and to enable the complete processor state to be saved and restored.

6.2.1 Kernel Mode

When `-sec_modes_option==true`, the secure kernel mode has the highest level of privilege and is the operating mode entered from [Reset](#). Access to complete machine state, including privileged instructions and privileged registers, is provided in secure kernel mode. When `-sec_modes_option==false`:

The normal kernel mode has the highest level of privilege and is the operating mode entered from [Reset](#). Access to complete machine state, including privileged instructions and privileged registers, is provided in kernel mode.

6.2.2 User Mode

The user mode has the lowest level of privilege and provides restricted access to privileged machine state. Any attempt to access privileged machine state, or to execute privileged instructions raises an exception.

6.2.3 Privilege Violations

This section describes the privileges available to the tasks running in user mode and kernel mode. [Table 6-1](#) gives an overview of the differences in privilege between the two modes.



Note

TLB operations are not supported on ARC EM family of processors.

Table 6-1 Overview of Privileges in Kernel and User Modes

Function	User Mode	Kernel Mode
Access to General Purpose Registers	All, except ILINK for which no access is allowed in user mode	Yes
Memory management / TLB controls	No	Yes
Cache management	No	Yes
Access to memory with ASID = User PID	By flag bits in Page Descriptor (PD)	By flag bits in Page Descriptor (PD)
Access to memory with ASID ≠ User PID	If global bit set	If global bit set
Unprivileged instructions	Yes	Yes
Privileged instructions	No	Yes

Table 6-1 Overview of Privileges in Kernel and User Modes (Continued)

Function	User Mode	Kernel Mode
Access to Basecase Auxiliary Registers	Only LP_START, LP_END, PC and STATUS32[ZNCV]	Yes
Access to extension auxiliary registers	JLI_BASE, LDI_BASE and EI_BASE	Yes
Build Configuration Registers	No	Yes
Timer access	No	Yes
TRAP_S n, SWI, SWI_S	Yes	Yes
Interrupt Enable, level selection	No	Yes
Extension instructions and state	If permitted by XPU	Yes

6.2.3.1 Privileged Instructions

All ARCV2 instructions are unprivileged unless specifically defined as privileged. When `-sec_modes_option==false`, the following instructions are privileged:

Privileged instructions are:

- [SLEEP](#)
- [RTIE](#)
- [CLRI](#)
- [SETI](#)

The following instructions are privileged when `DEBUG[UB]=0`:

- [BRK](#)
- [BRK_S](#) (see [BRK](#))

The [KFLAG](#) instruction is accessible in both user and kernel modes, but in kernel mode this instruction has amplified privilege and is able to modify certain protected bits in STATUS32.

[Table 6-2](#) describes the privilege access of instructions.

Table 6-2 Privilege Access of Instructions when the Secure-Mode Option is Configured

Instruction	Access Privilege
SLEEP	■ If SEC_STAT.NIC==0, the SLEEP instruction is allowed in the normal and secure kernel modes. However, in the normal kernel mode, you can only set the STATUS32.IE bit only; Writes to the STATUS32.E bit are ignored. ■ If SEC_STAT.NIC==1, the SLEEP instruction is allowed in the normal kernel and the secure kernel mode. You can set the STATUS32.IE and STATUS32.E fields. ■ Using the SLEEP instruction in the normal or secure user modes, raises an EV_PrivilegeV exception.
RTIE	Available only in the kernel modes.
CLRI	■ When SEC_STAT.NIC==0: - The CLRI instruction is allowed only in the secure kernel mode. - Using the CLRI instruction in any other mode raises a Privilege Violation exception (0x071000). ■ When SEC_STAT.NIC==1: - The CLRI instruction is available in the secure kernel and normal kernel modes. - Using the CLRI instruction in the secure or normal user mode raises a Privilege Violation exception (0x071020).
SETI	■ When SEC_STAT.NIC==0: - The SETI instruction is allowed only in the secure kernel mode - Using the SETI instruction in any other mode raises a Privilege Violation exception (0x071000). ■ When SEC_STAT.NIC==1: - The SETI instruction is allowed in the secure kernel and normal kernel modes. - Using the SETI instruction in the secure or normal user mode raises a Privilege Violation exception (0x071020).
BRK	This instruction is privileged when DEBUG[UB]=0
BRK_S	This instruction is privileged when DEBUG[UB]=0
FLAG	You can use the FLAG instruction to modify the STATUS32.H flag in the secure kernel mode only. If you attempt to set STATUS32.H to 1 in the normal kernel mode, a Privilege Violation is generated. Attempts to set the H bit using FLAG in both normal user mode and secure user mode are ignored.
KFLAG	You can use the KFLAG instruction to modify the STATUS32.H flag in the secure kernel mode only. If you attempt to set STATUS32.H to 1 in the normal kernel mode, a Privilege Violation is generated. Attempts to set the H bit using KFLAG in both normal user mode and secure user mode are ignored.
SFLAG	This instruction can be executed in either secure modes (kernel or user), execution of this instruction in normal user or kernel mode results in an EV_PrivilegeV (ECR = 0x071020) exception. This instruction is only available -sec_modes_option == true. If -sec_modes_option == false, and you use this instruction, an illegal instruction exception (0x020000) is raised.

Table 6-2 Privilege Access of Instructions when the Secure-Mode Option is Configured

Instruction	Access Privilege
WEVT	■ If SEC_STAT.NIC==0, the WEVT instruction is available in the normal and secure kernel modes. However, in the normal kernel mode, you can only set the STATUS32.IE bit; Writes to the STATUS32.E bit are ignored. ■ If SEC_STAT.NIC==1, the WEVT instruction is available in the normal kernel and the secure kernel mode. You can set the STATUS32.IE and STATUS32.E fields. ■ Using the WEVT instruction in the normal or secure user modes, raises an EV_PrivilegeV exception.

6.2.3.2 Privileged Registers

Access to the majority of general-purpose registers is not affected by the operating mode. Switching between the user and kernel modes does not affect the contents of general-purpose registers. No access is permitted to the ILINK register from user mode (see [Link Registers, ILINK \(r29\), BLINK \(r31\)](#)).

Illegal access from user mode to ILINK raises a Privilege Violation exception (see [Privilege Violation, Kernel/User Only Access](#)) and the cause is indicated in the exception cause register (see [Exception Cause Register, ECR](#)).

Moves to and from auxiliary registers are permitted in both the user and kernel modes. The auxiliary registers accessible in the user mode include the following:

- [Program Counter, PC](#)
- [Status Register, STATUS32](#) - ZNCV flags only
- [Loop Start Register, LP_START](#)
- [Loop End Register, LP_END](#) - where permitted by extension enables

The remaining auxiliary registers are accessible only in the normal kernel mode or the secure kernel mode.

6.2.4 Switching Between Operating Modes and Privilege Levels

The processor includes two privilege levels: user and kernel. Also, the processor includes two operating modes: secure and normal. The privileges and the operating modes are orthogonal to each other. The following sections explain the switching between the privilege levels and the operating modes.

6.2.4.1 Switching Between Privilege Levels

The processor may transition into a kernel mode when the following instructions are executed or the following events take place:

- [TRAP_S, SWI, SWI_S](#)
- Interrupt
- Exception
- [Reset](#) or Machine check exception; a Reset sets the core in the secure kernel mode.

The processor transitions from the kernel mode to user mode under the following conditions:

- Return from exception (see [Exception Exit](#)) - when the machine status register indicates that the last exception was taken from user mode.
- Return from interrupt (see [Regular Interrupts](#)) - when an interrupt is taken from a user mode, the return from that interrupt switches the mode to user mode. When the processor is returning from an interrupt and another enabled interrupt is pending, the processor avoids restoring the registers on interrupt exit by comparing the priority level of the exiting interrupt to the highest pending priority interrupt. For more information, see [“Returning from Regular Interrupts”](#).

Exception and interrupt handlers may choose to adjust the values in their return address (see [Exception Return Address](#), [ERET](#), [Link Registers](#), [ILINK \(r29\)](#), [BLINK \(r31\)](#)) and status link registers (see [Exception Secure Status Register](#), [ERSEC_STAT](#), [Exception Return Status](#), [ERSTATUS](#), [Status Register Priority 0](#), [STATUS32_P0](#)) and the [AUX_IRQ_ACT \(Active Interrupts Register, AUX_IRQ_ACT\)](#) register to jump to a kernel mode or user mode task when clearing the interrupt-active or exception-active bits in the status register.

The FLAG instruction cannot be used to change the user-mode or kernel-mode state of the processor, but a KFLAG instruction executed in kernel mode can place the processor in exception-handling mode, or at the normal (non-exception, non-interrupt) mode, or set the interrupt threshold (E[3:0] bits of STATUS32).

6.2.4.2 Software-Initiated Switching from Secure to Normal Operating Mode

For calls from secure mode to normal mode, the processor can detect that a branch-and-link or jump-and-link is executing in secure mode and the target of the branch or jump is in a normal memory region. The processor then performs the following steps:

1. Sets the BLINK register to 0x1 to indicate that the call was from secure to normal mode.
2. Saves the return address to the secure stack.
3. Saves the SEC_STAT register to the secure stack.
4. Branches to the target address, automatically switching to normal mode when fetching from the target address.

For direct software calls to normal region from a secure region, registers with secure shadow or banked values (such as the stack register) are automatically switched from secure-mode to normal-mode registers. No special handling is provided for the register file, so the calling function must take care to leave only the required data in the register file prior to making the call, to prevent information leaking from the secure context. If the call is due to a compiler-inserted support-function call then such cleanup must be handled by the generated code.

When executing a `jump [BLINK]` to return from a normal-mode function, a value of 0x1 in the BLINK register causes the processor to pop the actual return address from the secure stack.

6.2.4.3 Software-Initiated Switching from Normal to Secure Operating Mode

Switching from a normal to the secure mode is implicitly done when program flow jumps to an instruction in the secure memory. To prevent arbitrary jumps into secure code, only jumps into controlled entry points are allowed. This design is implemented using the SJLI instruction and a secure jump table. The return address in normal mode is stored in the BLINK register as usual. Like for calls from secure to normal, the programmer should take care on returning, to ensure that no sensitive data is left in general-purpose registers, as these registers are not automatically cleared.

If branches and jumps in normal mode are executed into the secure address space without using the SJLI instruction, an Instruction Error exception is triggered.

If instruction execution linearly enters the secure address space, an Instruction Error exception is triggered.

If either the secure jump table or the target of the location in the secure jump table are not located in the secure address space, an EV_MachineCheck exception is triggered when the call is issued.

6.2.4.4 Hardware-Triggered Switching Between Secure and Normal Operating Modes

When a switch from normal mode to secure mode is triggered by hardware (interrupt or exception where the handler is in secure mode), no special handling is needed. However, when the core is in secure mode and an interrupt or exception is triggered that must be handled in normal mode, the secure context is hidden prior to switching to the normal-mode context: the full secure context register set is pushed to the secure stack and all the registers in the secure context are set to zero prior to switching to the normal code.

6.2.4.5 Function and Interrupt Returns Across Modes

- To allow a function return from normal mode to secure mode, following criteria must be met:
 - SEC_STAT.NSRT == 0
 - SEC_STAT.NSRTU==STATUS32.U
- For an interrupt to return from a normal mode to a secure mode, the following criteria must be met:
 - SEC_STAT.NSRT == 1
 - Additionally, when the interrupt return is not to a nested-interrupt context (if only one bit is set in AUX_IRQ_ACT.ACT[15:0]), the return should be to the correct user or kernel privilege level:
SEC_STAT.NSRU ==AUX_IRQ_ACT.U
- The return from a secure mode to normal mode from an IRQ is determined by the SEC_STAT.IRM bit. The return mode from an exception is determined by the ERSEC_STAT.ERM bit. This bit can be programmed in the secure kernel mode. However, in the normal kernel mode, this bit cannot be programmed.
- Whether a mode-changing micro-operation sequence is pending while a delay slot instruction is executing is indicated by the SEC_STAT.SUE bit. On exception return, ERSEC_STAT.SUE is restored to the SEC_STAT.SUE bit and triggers the mode switch micro-op sequence after the delay slot or immediately. You can cancel the pending micro-op sequence before an exception return by clearing the ERSEC_STAT.SUE bit.

6.2.5 Register Replication

When `-sec_modes_option==true`, the following registers are duplicated and automatically switched when the core switches between the secure and normal modes.



The replicated registers are internal to the core; these registers are not visible in the auxiliary register space.

- Loop Start & Loop End
- JLI_BASE/LDI_BASE/EI_BASE

- BTA
- Loop count
- Stack pointer registers: AUX_USER_SP, AUX_KERNEL_SP, AUX_SEC_U_SP, and AUX_SEC_K_SP
- Stack checking registers: USTACK_BASE, USTACK_TOP, KSTACK_BASE, KSTACK_TOP, S_USTACK_BASE, S_USTACK_TOP, S_KSTACK_BASE, S_KSTACK_TOP

**Note**

The accumulator registers, if present, must be included in the hardware register stack/zero sequences.

6.3 Interrupts

The ARCV2 interrupt unit has 16 allocated exceptions associated with vectors 0 to 15 and 240 interrupts associated with vectors 16 to 255. The interrupt unit is optional in the ARCV2-based processors. When building a processor, you can configure the processor to include an interrupt unit. The ARCV2 interrupt unit is highly programmable. The ARCV2 interrupt unit supports the following interrupt types:

- Fixed core interrupts: internal interrupts with preassigned vector positions
- Shared common interrupts: external interrupts distributed to each core by the IDU
- Safety error interrupt: non-preassigned internal interrupt
- External interrupts — available as core specific (private) interrupt input pins to the core
- Software interrupts — triggered by software only

This section explains the following topics:

- [Interrupt Unit Features](#)
- [Interrupt Unit Configuration](#)
- [Interrupt Unit Programming](#)
- [Interrupt Handling](#)

6.3.1 Interrupt Unit Features

ARCV2 interrupt unit has the following interrupt specifications:

- Support for up to 240 interrupts
 - User configurable from 0 to 240
 - Level sensitive or pulse sensitive
- The ARCV2 ISA permits up to 16 interrupt levels; individual ARCV2-based processors may implement fewer levels. For precise information for your ARCV2-based processor, see the processor *Databook*.
 - Programmable from 0 (highest priority) to 15 (lowest priority)
- Support for configuring an interrupt as a secure interrupt or a normal interrupt.

- The priority of each interrupt can be programmed individually by software
- Interrupts can be preempted by higher-priority interrupts
 - Optionally, highest priority level 0 interrupts can be configured as 'Fast Interrupts'
 - Optional second core register bank for use with [Fast Interrupts](#) option to minimize interrupt service latency by minimizing the time needed for context saving
- Automatic save and restore of selected registers on interrupt entry and exit for fast context switch
- User context saved to user or kernel stack, under program control
- Software can set a priority level threshold in STATUS32.E that must be met for an interrupt request to interrupt or wake the processor
- Minimal interrupt / wake-up logic clocked in sleep state
 - Interrupt prioritization logic is purely combinational
- Any Interrupt can be triggered by software

6.3.2 Interrupt Unit Configuration

You can configure the interrupt controller using the options in [Table 6-3](#):

Table 6-3 Interrupt Configuration Options

Configuration Option	Range	Description
has_interrupts	false, true	Indicates if the processor configuration includes the interrupt unit. false—indicates that the interrupt unit is not included. true—indicates that the interrupt unit is included.
number_of_interrupts	0 to 240 interrupts	Defines the total number of interrupts in the interrupt controller.
number_of_levels	1 to 16	Defines the number of interrupt priority levels (0 to 15) in the interrupt controller.
external_interrupts	up to 240	Defines the number of external interrupts. External interrupts are interrupt pins that are available for you to connect to your components. Additional external interrupts are available when ARConnect IDU is configured.
firq_option	false, true	Indicates whether the fast interrupt option is enabled. false: indicates that the fast interrupt option is disabled. true: indicates that the fast interrupt option is enabled.



Note

For more information about the Configuration options for ARCV2-based processors, see [Configurability](#).

6.3.2.1 Interrupt Configuration Constraints

The ARCV2 ISA includes the following interrupts:

Table 6-4 Interrupt Allocation

Signal	Description
irq16_a	Timer 0
irq17_a	Timer 1
irq18_a	Watchdog timer
irq19_a	Safety interrupt when safety is configured
irq20_a	Secure timer 0 when secure timers are configured; If secure pipeline (has_secure) is not configured in the processor, this interrupt is assigned to micro-DMA controller (if configured)
irq21_a	Secure timer 1 when secure timer 1 is configured
irq22_a	This interrupt is assigned for single-bit error interrupts when SECDED is configured in ECC protection with secure enhancement package (<code>(!err_prot_option) pipe_prot_secDED</code>). Otherwise, the micro-DMA controller interrupts are assigned starting with this interrupt number. Single-bit error occurs in any of the following: ■ memory component including CCM and Cache ■ core register file and PC ■ MPU auxiliary registers ■ STATUS32 Note: For single-bit errors, this interrupt is a level-sensitive interrupt and it will not be cleared automatically after acknowledgment. You can clear this interrupt by writing to the specific aux register in the interrupt handler.
irq23_a	■ micro-DMA controller interrupts are assigned starting with this interrupt number if irq22_a is occupied by the single-bit ECC error interrupt.

**Note**

If the DMA channels are configured to raise interrupts, the interrupt vectors are assigned as follows:

- Single-Internal: Two signals for all DMA channels: one for error and one for completion.
- Single-External: Two signals for all DMA channels: one for error and one for completion. The external interrupts are assigned as a block to the lowest available unused interrupt vectors.
- Multiple-Internal: Two signals per DMA channel. The number of interrupts assigned is (2*DMAC_CHANNELS). These interrupts are assigned as a block starting with the lowest available unused internal interrupts (only after irq20).
- Multiple-External: Two signals for each DMA channel: one for error and one for completion. The number of interrupts assigned is (2*DMAC_CHANNELS). The external interrupts are assigned as a block to the lowest available unused interrupt vectors.

- External interrupts: core specific (private) interrupt
 - Number of external interrupt pins that you can connect to your components. The maximum number of private (core-specific) external pins can be computed as follows: (Number of interrupts – fixed core interrupts – shared common interrupts – safety error interrupt). External interrupts are assigned after all internal sources (including Safety Error Interrupt) are assigned, allocating the lowest available unused interrupt vectors first. For example, if only TIMER1 is present, external interrupts are assigned vectors 16, 18, 19, and so on.
- Software-only triggered interrupts
 - Software-only triggered interrupts, if any, are assigned vectors starting just above the external interrupts.

The interrupt configuration options selected in a particular build are reflected in the read-only [Interrupt Build Configuration Register, IRQ_BUILD](#) register.

6.3.2.2 Interrupt Vectors

Each vector is a 32 bit address pointing to the appropriate handler for the exception or interrupt. The vectors are stored in the native endianness.

[Table 6-5](#) lists the vector offsets. One 32-bit word is reserved for each interrupt line to allow room for vector address. Vectors are fetched in instruction space and thus may be present in ICCM, Instruction Cache, or main memory accessed by instruction fetch logic.

When an interrupt or exception occurs, the processor obtains the address of the entry point of the interrupt or exception handler from a vector table in memory. This vector table contains one 32-bit entry point for each interrupt or exception, and is indexed by the exception or interrupt vector number.

The INT_VECTOR_BASE register (see [Interrupt Vector Base Register, INT_VECTOR_BASE](#)) indicates the base address of the interrupt vectors. This register can be read at any time to determine the start of the interrupt vectors, and can be used to change the base of the interrupt vectors during program execution.

When a normal interrupt is triggered, the INT_VECTOR_BASE register (see [Interrupt Vector Base Register, INT_VECTOR_BASE](#)) indicates the base address of the interrupt vectors. This register can be read at any time to determine the start of the normal interrupt vectors, and can be used to change the base of the normal interrupt vectors during program execution.

When `-sec_modes_option==true`, the processor supports secure interrupts. When a secure interrupt is triggered, the `INT_VECTOR_BASE_S` register (see [Secure Interrupt Vector Base Register, INT_VECTOR_BASE_S](#)) indicates the base address of the secure interrupt vectors. This register can be read at any time to determine the start of the secure interrupt vectors, and can be used to change the base of the secure interrupt vectors during program execution.

6.3.2.3 Interrupt Vector Base Address Configuration

When `-sec_modes_option==true`, on [Reset](#), the secure interrupt vector base configuration register, initializes the base address of the secure interrupt vectors. This base address is loaded into the secure interrupt vector base address register, `INT_VECTOR_BASE_S`, on Reset.

The normal interrupt vector base configuration register (see [Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD](#)), initializes the base address of the interrupt vectors. This base address is loaded into the normal vector base address register (see [Interrupt Vector Base Register, INT_VECTOR_BASE](#)), on Reset.

During program execution, you can adjust the base address of the normal interrupt vector table by modifying the [Interrupt Vector Base Register, INT_VECTOR_BASE](#).

On [Reset](#), the interrupt vector base configuration register (see [Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD](#)), initializes the base address of the interrupt vectors. This base address is also loaded into the interrupt vector base address register (see [Interrupt Vector Base Register, INT_VECTOR_BASE](#)), on [Reset](#).

During program execution, you can adjust the base address of the interrupt vector table by modifying the [Interrupt Vector Base Register, INT_VECTOR_BASE](#).

6.3.3 Interrupt Unit Programming

The interrupt unit allows programming of certain parameters.

When a configuration has no interrupt controller, the interrupt-related registers are not present and access to nonexistent interrupt registers raises an Illegal Instruction exception in all modes.

6.3.3.1 Program the Interrupt Unit

The following are some of the steps that you can follow to program the interrupt unit:

1. Disable the global interrupt unit in the processor by setting the IE bit to 0 in the STATUS32 register.
2. Select an interrupt register bank using [IRQ_SELECT](#): Write the interrupt number to the `IRQ_SELECT` register. The corresponding interrupt register bank for the selected interrupt is enabled. The following are the interrupt registers you must program.
 - a. [IRQ_PRIORITY](#): Program the priority of the selected interrupt by writing the priority (0 to `(NUMBER_OF_LEVELS-1)`) to the `IRQ_PRIORITY` register. By default, `IRQ_PRIORITY` is set to 0. Program the `IRQ_PRIORITY.S` bit to configure an interrupt as a secure interrupt or normal interrupt.
 - b. [IRQ_TRIGGER](#): Program the interrupt as level-sensitive or pulse-sensitive by writing to this register. By default, the interrupts are level-sensitive. Write 0 to this register to configure the interrupt as level-sensitive. Write 1 to this register to configure the interrupt as pulse-sensitive.
 - c. [IRQ_PULSE_CANCEL](#): Write 1 to this register to cancel the pulse-sensitive interrupt. If you write 0 to this register, it is always ignored.

- d. [IRQ_ENABLE](#): Write 1 to the `IRQ_ENABLE` register to enable the interrupt. By default, the `IRQ_ENABLE` register is set to 1.
 - e. [IRQ_PENDING](#): This register is read-only. This register indicates if the selected interrupt is pending.
 - f. [IRQ_STATUS](#): This register is read-only. This register summarizes the status of all the interrupt bank registers for the interrupt selected using `IRQ_SELECT`.
3. Set interrupt priority threshold of the processor by writing to `E[3:0]` bits in `STATUS32`.
 4. Enable global interrupt unit in the processor by setting the `IE` bit to 1 in the `STATUS32` register.
 5. For software debugging and simulation: Write the interrupt number you want to trigger to the `AUX_IRQ_HINT` register. This register triggers the selected interrupt. The interrupt unit services this interrupt based on interrupt prioritization and preemption. For more information, see [Interrupt Prioritization and Preemption](#).

The following sections explain in detail how you can program the ARCV2 interrupt unit. Ensure that you have the necessary privileges to program the interrupt unit.

6.3.3.2 Enabling Interrupts

You can enable interrupts by setting the `IE` bit in `STATUS32` ([Status Register, STATUS32](#)) using the [SETI](#) instruction or when the [RTIE](#) instruction restores `STATUS32`.

6.3.3.3 Disabling Interrupts

You can disable interrupts by using the [CLRI](#) instruction that always forces the `STATUS32.IE` bit to 0, disabling interrupts.

6.3.3.4 Pending Interrupts

The [Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING](#), provides software visibility of all priority levels at which an interrupt is pending, including levels below the currently-active interrupt.

Read the `IRQ_PRIORITY_PENDING` register to know the pending interrupts (secure or normal) of all priority levels. Only interrupts matching the current privilege level are indicated, that is, pending secure interrupts are ignored when the register is read from normal mode.

For each interrupt, the [Interrupt Pending Register, IRQ_PENDING](#) register indicates its pending status. A pending interrupt can be canceled or reset based on the interrupt sensitivity ([Interrupt Sensitivity Level Programming](#)).

6.3.3.5 Exception and Interrupt Priority

Exceptions, like [Reset](#) and [Illegal Instruction](#), have a higher priority than interrupts. Interrupts have a higher priority than the synchronous program exceptions, such as traps, and arithmetic exceptions. This prioritization limits the worst-case interrupt response time of the processor. For detailed list of interrupt and exception priorities, see [Exception Types and Priorities](#).

The dynamic interrupt priority model of ARCV2 ISA provides up to 16 levels of priority, and allows each configured interrupt to have its interrupt priority set dynamically under software control.

6.3.3.6 Interrupt Prioritization and Preemption

In ARCV2-based processors, Priority 0 (P0) is the highest interrupt priority, and Priority 15 (P15) is the lowest. Current operating interrupt priority defines how the processor reacts when servicing an interrupt or an exception and a new interrupt is activated.

The E[3:0] field of STATUS32 defines the interrupt priority threshold that is enabled in the processor. Interrupts of lower priority than defined by E[3:0] are not serviced. For example, set E = 4 to enable only interrupts at priority 4 (P4) or higher priority than P4 (P0, P1, P2, and P3). Similarly, set E = 15 to enable interrupts at all priority levels.

An interrupt number is a unique integer value, in the range [16,255], that identifies each interrupt. It is also the index within the vector table at which the vector for that interrupt is located.

A priority is an integer in the range [0, NUMBER_OF_LEVELS]. The priority of an interrupt is set in the IRQ_PRIORITY register associated with every interrupt and is in the range [0 to NUMBER_OF_LEVELS-1].

An enabled interrupt is any interrupt for which the corresponding IRQ_ENABLE.E == 1. An active interrupt is defined as an enabled interrupt that is also one of the following:

1. An external level-sensitive interrupt with an asserted interrupt request signal, or
2. An external pulse-sensitive interrupt with an outstanding interrupt request, or
3. An internal software-triggered interrupt with an interrupt number equal to the value in AUX_IRQ_HINT.

The current operating interrupt priority for a processor is determined by the AUX_IRQ_ACT register. The bit position, 0 - 15, of the least-significant 1 in the lower 16 bits of the AUX_IRQ_ACT register is the current operating interrupt priority. If the lower 16 bits of the AUX_IRQ_ACT register are zero, it indicates that the processor is not currently handling an interrupt.

A processor is interrupted if the following conditions are met:

1. The processor is not halted (STATUS32.H == 0), and
2. Interrupts are globally enabled (STATUS32.IE == 1), and
3. The processor is not currently handling an exception (STATUS32.AE == 0), and
4. If there exists an active interrupt with priority that is higher than (that is, numerically less than) the current operating interrupt priority, and the active interrupt has the same priority or higher than the interrupt preemption threshold (that is, numerically less than or equal to) specified by the STATUS32.E bits.



Hint

For more information about the behavior and timing of interrupts, see the ARCV2-based processor Databook of the core you are using.

When these conditions are met, the active interrupt with the highest priority is serviced. If more than one such interrupts are active, the interrupt with the lowest interrupt vector number is serviced.

If the processor is in a sleeping state, it is awakened before the processor is interrupted.

6.3.3.7 Banked Interrupt Registers



Note

When `-sec_modes_option==true`, banked interrupt registers are not supported.

An interrupt controller supporting M interrupts ($1 \leq M \leq 240$), has M banks of interrupt-specific auxiliary registers, selected by [Interrupt Select, IRQ_SELECT](#). A processor must write to `IRQ_SELECT` before reading or writing to one of the banked registers.

The following are the set of registers that are replicated for each interrupt selected by the `IRQ_SELECT` register:

- [Interrupt Priority Register, IRQ_PRIORITY](#) – This register defines the priority level of the interrupt selected by the `IRQ_SELECT` register. A priority register with a value $\geq N$ confers priority $N-1$; where N is the number of interrupt priorities. If multiple asserted interrupts are programmed at the same priority level, the interrupt with the lowest number is given the highest priority.
- [Interrupt Enable Register, IRQ_ENABLE](#) – When this register is set to 1, the interrupt selected by the `IRQ_SELECT` register is enabled.
- [Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL](#) – Pulse-triggered interrupts are generally auto clearing. If you choose not to take a pulse-triggered interrupt, you can use this banked register to clear that pulse-triggered interrupt for the IRQ selected by the `IRQ_SELECT` register. This register is selected by the value in the `IRQ_SELECT` register. This register is available only for external interrupts.
- [Interrupt Trigger Register, IRQ_TRIGGER](#) – This register allows software to program whether the interrupt is level-sensitive or pulse-sensitive by writing 0 or 1 respectively. This register is available only for external interrupts.
- [Interrupt Pending Register, IRQ_PENDING](#) – When this register is set to 1, it indicates that the interrupt, selected by the `IRQ_SELECT` register, is pending.
- [Interrupt Status Register, IRQ_STATUS](#) – This register allows software to read the combined values in `IRQ_PRIORITY`, `IRQ_PENDING`, `IRQ_ENABLE` and `IRQ_TRIGGER` using a single status register for the selected interrupt. This register has $n+3$ bits.

There is a single `IRQ_PRIORITY_PENDING[15:0]` register, which provides visibility of whether there is any interrupt pending at each priority level.

Reading or writing to a banked interrupt register is a two-stage process. Interrupts have to be disabled during this process to avoid the `IRQ_SELECT` register being modified by a separate thread. For banked interrupt registers that are unimplemented, writes are ignored, and reads return zero.

[Example 6-1](#) explains how to set the interrupt given by register `R1` to the priority given by register `R2`. Interrupt status is saved and restored using `R0`.

Example 6-1 Setting Interrupt Priority or Enable (STATUS32.IE) of an Interrupt

```
CLRI  R0                ;;save interrupt enable and then disable
SR    R1, [IRQ_SELECT]  ;;R1 provides the interrupt number
SR    R2, [IRQ_PRIORITY] ;;R2 provides the interrupt priority
SETI  R0                ;;restore previous interrupt enable
```

**Note**

The CLRI and SETI instructions are serializing instructions.

6.3.3.8 Priority Level Programming

The priority level of each interrupt is programmable. Interrupt priority is configured by setting the priority level in the [Interrupt Priority Register, IRQ_PRIORITY](#) register of the interrupt selected by the [Interrupt Select, IRQ_SELECT](#) register.

[Example 6-1](#) explains how to set the interrupt given by register R1 to the priority given by register R2. Interrupt status is saved and restored using R0.

6.3.3.9 Interrupt Sensitivity Level Programming

Each interrupt can be programmed to be either pulse-sensitive or level-sensitive.

An interrupting device, set to pulse-sensitive interrupts, must assert the interrupt line only once, and then de-assert the interrupt line. For pulse-sensitive interrupts, at most one interrupt is generated for each positive transition of an interrupt line. You can clear the pending interrupt by writing 1 to the corresponding [IRQ_PULSE_CANCEL](#) register.

An interrupting device, set to level-sensitive interrupt, must assert and hold the interrupt line until instructed to de-assert the interrupt line by the corresponding interrupt service routine.

**Hint**

For more information about the behavior and timing of interrupts, see the ARCV2-based processor Databook of the core you are using.

All interrupts are level-sensitive by default, but can be individually changed to pulse-sensitive by writing the appropriate banked [Interrupt Trigger Register, IRQ_TRIGGER](#).

This banked register allows software to set and examine the current level or pulse trigger setting for the IRQ selected by the [Interrupt Select, IRQ_SELECT](#) register.

Writing 0 to the T field of [Interrupt Trigger Register, IRQ_TRIGGER](#) sets the sensitivity of the interrupt selected by [Interrupt Select, IRQ_SELECT](#) to be level sensitive. If the T field is set to 1, the associated interrupt will be pending for one cycle after a transition from 0 to 1 is detected on the interrupt line (sampled at CPU clock frequency). The interrupt remains pending until it is cleared by writing 1 to the corresponding [Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL](#) register.

6.3.3.10 Canceling Pulse Triggered Interrupts

The write-only 32-bit register (see [Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL](#)) allows the operating system to clear a pulse-triggered interrupt for the IRQ selected by the [IRQ_SELECT](#). Writing pulse-canceling value 1 to a non-level sensitive interrupt clears any saved interrupt. A write of pulse-canceling value 0 is always ignored.

A write of pulse-canceling value 1 to a non-level sensitive interrupt clears any saved interrupt.

6.3.3.11 Software-Triggered Interrupts

In addition to the SWI instruction, the interrupt system allows software to generate a specific interrupt by writing to the software interrupt trigger register (see [Software Interrupt Trigger, AUX_IRQ_HINT](#)). All interrupts can be generated through the AUX_IRQ_HINT register (see [Software Interrupt Trigger, AUX_IRQ_HINT](#)). The [Software Interrupt Trigger, AUX_IRQ_HINT](#) register can be written by a program running on the processor or from the host.

When you write to the AUX_IRQ_HINT register in the normal mode, you can only generate the normal interrupts. Attempts to generate secure interrupts from the normal interrupts are ignored. You can generate secure interrupts from the secure kernel mode only.

The software-triggered interrupt mechanism can be used even if there are no associated interrupts connected to the ARCV2-based processor. Writing AUX_IRQ_HINT (see [Software Interrupt Trigger, AUX_IRQ_HINT](#)) with a value for which there is no internal or external interrupt implemented (such as 0), clears any pending software triggered interrupt.

The delay between writing to the AUX_IRQ_HINT register (see [Software Interrupt Trigger, AUX_IRQ_HINT](#)) and the interrupt being taken is implementation specific.

6.3.3.12 Interrupt Cause Registers

The ICAUSE ([Interrupt Cause Registers, ICAUSE](#)) register is banked and there are N copies, one corresponding to each priority level. Each banked ICAUSE register records the number of an interrupt that is taken at the register's corresponding priority level. When the AUX_IRQ_ACT register is non-zero, reading the ICAUSE register returns the interrupt number of the highest priority active interrupt (the lowest bit set in the AUX_IRQ_ACT register). When AUX_IRQ_ACT is zero (no active interrupts), the ICAUSE register read value is undefined. The interrupt cause register, ICAUSE is not affected when returning from an interrupt.

If a normal interrupt is yet to be taken, reading this register returns 0x00000001. If secure interrupts are taken, and you read the ICAUSE register corresponding to a particular priority level in the normal kernel mode, the last normal interrupt number that was taken at this priority level is returned. A read of an ICAUSE register returns the interrupt number irrespective of whether the interrupt is a secure interrupt or not.

6.3.4 Interrupt Handling

The ARCV2-based processors handles interrupts based on the following scenarios:

Fast interrupts scenario-- A scenario where on interrupt entry, the processor stores only partial user context (PC and STATUS32 registers) for the highest priority interrupts, P0. The purpose of fast interrupt option is to avoid memory transactions in the entry and exit sequences for interrupts at level P0.

Regular interrupts scenario-- A scenario where the selected user context is saved and restored from the stack.

6.3.4.1 Break on Interrupt or Exception Vector

If a breakpoint is enabled on an interrupt or exception vector fetch, using the DEBUGI register and the least-significant bit of the exception vector, the CPU halts with PC set to the address of the selected vector entry. Before the CPU halts, on a vector breakpoint, it completes the context save sequence required for that interrupt or exception. This design allows the debugger to restart after the halt by simply assigning the vector value to PC and restarting the CPU.

6.3.4.2 Fast Interrupts

You can configure the interrupt architecture so that fast interrupts, interrupts with the highest priority level P0, save only PC and STATUS32 registers. The purpose of the fast interrupt option is to avoid memory transactions in the entry and exit sequences for interrupts at level P0. An optional second core register bank may be used so that these registers need not be saved or restored from a stack when servicing a fast interrupt. Use the following option to configure fast interrupts:

`FIRQ_OPTION`

The following is the behavior of fast interrupt option:

When `FIRQ_OPTION` is 1, the mechanism to push and pop PC, STATUS32, and general purpose registers to the stack on interrupt entry or exit is disabled for level P0 interrupts. When the fast interrupt option is enabled, the STATUS32 register is stored to STATUS32_P0 and the PC register is stored to ILINK during the lifetime of a P0 interrupt handler.

When the `FIRQ_OPTION` is 0, the STATUS32_P0 register is read as zero and ignored on writes, and P0 priority level interrupts behave the same way as other priority levels; context is saved and restored from a stack in memory. Also, when the `FIRQ_OPTION` is 0, the ILINK register is used for temporary storage by the hardware during interrupt entry and exit, and is not usable as a general purpose register.

When `FIRQ_OPTION` is 1 and `RGF_NUM_BANKS` $\geq$ 2, the interrupt entry sequence sets the register bank number STATUS32.RB to 1, the second register bank, when entering a priority P0 interrupt. Similarly, on P0 interrupt exit, the processor returns STATUS32.RB to its previous value by restoring STATUS32.RB from STATUS32_P0. This configuration causes an automatic switch of the duplicated register banks, allowing all P0 interrupts to use a private subset of registers.



Note

`firq_option==true` does not necessarily require that `RGF_NUM_BANKS > 1`.

6.3.4.2.1 Context Save and Restore during Fast Interrupts

When a fast interrupt (P0) occurs, the appropriate link register is loaded with the value of next PC, the associated STATUS32_P0 status save register is updated with the status register (see [Status Register, STATUS32](#)), and the PC is loaded with the relevant address for servicing the interrupt.

The ILINK register is not duplicated across register banks. Reads and Writes to ILINK are performed as if {STATUS32,DEBUGI}.RB was equal to zero regardless of the current machine state.

The use of ILINK and STATUS32_P0 are associated with the highest priority level, P0, and exist as such only when `FIRQ_OPTION = 1`; the ILINK and STATUS32_P0 registers are used to automatically save the PC and STATUS32 when a level P0 interrupt is taken. If `FIRQ_OPTION = 0`, ILINK, r29 is used for temporary storage by the hardware during interrupt entry and exit, and is not usable as a general purpose register; and, STATUS32_P0 is read as zero and ignored on writes.

The PC and STATUS32 (along with other CPU state) are saved to a user or kernel stack in memory when taking non-P0 interrupts (when `firq_option==true`) or for all interrupts (when `firq_option==false`).

6.3.4.2.2 Fast Interrupt Entry

The following example explains the sequence of a fast interrupt entry. To understand the pseudo code, refer to the following notation:

- P is the processor's current operating priority defined by the index of least significant bit set (with value of 1) in AUX_IRQ_ACT.ACTIVE. P is 16 if no interrupts are active.
- Q is the index of the least significant bit set (with a value of 1) in AUX_IRQ_ACT.ACTIVE when bit P is cleared. Q is 16 if P is the only bit set to 1 in AUX_IRQ_ACT.ACTIVE.
- PI is the highest priority pending and enabled interrupt, and its priority is W and W=0.

Figure 6-1 Fast Interrupt Entry

```

EnterFastInterrupt (PI) {
    // Assert ((P != 0) && (STATUS32.IE) && (STATUS32.E ? 1)
    // && (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
    ILINK = PC;
    STATUS32_P0 = STATUS32;
    // save user-mode flag
    if (AUX_IRQ_ACT.ACTIVE == 0) {
        AUX_IRQ_ACT.U = STATUS32.U
    }
    AUX_IRQ_ACT.ACTIVE[W] = 1;
    if (RGF_NUM_BANKS > 1) {
        STATUS32.{RB=1}
    }
    // switch to kernel SP if previously in user thread
    if (STATUS32.U) {
        AEX R28, [AUX_USER_SP]
    }
    STATUS32.{Z = U; U = 0; L = 1; ES = 0; DZ = 0; DE = 0;}
    AcknowledgeInterrupt (PI);
    Jump (InterruptVector (PI));
}

```

6.3.4.2.3 Fast Interrupt Exit

The following pseudo-code explains the sequence of a fast interrupt exit:

Figure 6-2 Fast Interrupt Exit

```

ReturnFromFastInterrupt (PI, W) {
    // Assert ((P == 0) && (STATUS32.AE == 0))
    //if another fast interrupt is pending and enabled
    if ((W == 0) && (W ≤ STATUS32.E) && (STATUS32.IE == 1)) {
        //acknowledge interrupt
        AcknowledgeInterrupt (PI);
        //jump to another fast interrupt vector
        Jump (InterruptVector (PI));
    }
    //else restore context and return from interrupt
    else
    {
        //clear the currently-active interrupt
        AUX_IRQ_ACT.ACTIVE[P] = 0;
        //if (returning to a user thread)
        if ((AUX_IRQ_ACT.ACTIVE == 0) && AUX_IRQ_ACT.U) {

```

```

        //restore user SP
        AEX R28, [AUX_USER_SP];
        AUX_IRQ_ACT.U = 0;
    }
    //restore STATUS32
    STATUS32 = STATUS32_P0;
    //jump back to outer context
    Jump(ILINK);
}
}

```

6.3.4.3 Regular Interrupts

This section explains the handling of all interrupts when fast interrupt is disabled. The following section is also valid for all interrupts except P0 when fast interrupt is enabled.

6.3.4.3.1 Context Save and Restore for Regular Interrupts

The behavior of interrupt entry and exit is programmed through the `AUX_IRQ_CTRL` register. This design allows the kernel to define how much of the register context is saved automatically.

The saving and restoring of non-critical state for regular (non-FIRQ) interrupts is programmable. The auxiliary register, [Interrupt Context Saving Control Register, `AUX\_IRQ\_CTRL`](#), is used to control the automatic save and restore of processor state on interrupt entry and exit for non-fast interrupts.

The `ILINK` register may be modified on entry to an interrupt of any level, and may also be modified on exit from an interrupt at any level. Therefore, `ILINK` cannot be relied upon to retain its value unless executing within a level 0 interrupt handler. So, `PC` and `STATUS32` are always saved to the stack, as they are modified implicitly by the interrupt entry mechanism. `PC` and `STATUS32` are referred to as 'essential state'. The number of general purpose register pairs to save and restore can be programmed for non-fast interrupts to be any even number of baseline general purpose registers in the range 0 to 16, starting from `{R0, R1}`, and going up to `{R30, R31}`.

The `AUX_IRQ_CTRL.NR` field defines the number of general-purpose register pairs saved or restored from the stack on interrupt entry and exit. The register value is between 0 and 16 inclusive. The hardware save and restore mechanism ignores the `AUX_IRQ_CTRL.B` bit if $NR \geq 15$, as `BLINK` is saved as one of the general purpose registers in that case. If $AUX_IRQ_CTRL.NR < 15$, `BLINK` is saved and restored only if `AUX_IRQ_CTRL.B == 1`.

When the primary register bank is configured as 16 entries, the `AUX_IRQ_CTRL.NR` field must be between 0 and 8 inclusive. The set of registers saved and restored is `AUX_IRQ_CTRL.NR` pairs of the lowest numbered registers implemented as specified in [Table 5-3](#), Current ABI Register Usage. The hardware save and restore mechanism ignores the `AUX_IRQ_CTRL.B` bit if $NR \geq 8$, as `BLINK` is saved as one of the general purpose registers in that case. If $AUX_IRQ_CTRL.NR < 8$, `BLINK` is saved and restored only if `AUX_IRQ_CTRL.B == 1`.

The `AUX_IRQ_CTRL.L` field determines whether to save and restore `LP_COUNT`, `LP_START` and `LP_END` when entering or exiting an interrupt. If `JLI_BASE`, `EI_BASE` and `LDI_BASE` registers are configured (`code_density_option==true`) then those registers are also saved and restored when the `AUX_IRQ_CTRL.LP` field is set to 1.

If `AUX_IRQ_CTRL.U == 1` on interrupt entry, the saved registers are pushed to the current stack. The `STATUS32.S==1` bit determines whether the following operations happen in the secure mode or the normal mode:

- If `STATUS32.U == 1`, the user or kernel stack pointers are exchanged; Therefore, if `AUX_IRQ_CTRL.U == 1`, registers are pushed to the stack of the current user stack when a user thread is interrupted, or to the kernel stack when a kernel thread or interrupt handler is interrupted.
- If `AUX_IRQ_CTRL.U == 0` on interrupt entry, the saved registers are saved to the kernel stack regardless of operating mode and current interrupt level.
- When a user thread is interrupted (or an exception occurs), and `AUX_IRQ_CTRL.U == 1`, the user and kernel stack pointers are switched automatically after registers have been saved to the user stack. On return to a user thread, from either an interrupt or an exception, when `AUX_IRQ_CTRL.U == 1`, the kernel and user stack pointers are switched automatically before core registers are restored to the user stack. However, when `AUX_IRQ_CTRL.U == 0`, any stack pointer switching takes place before state is pushed, and after state is popped. In this case, the kernel stack is used.

**Note**

When the core is configured with fast interrupts (`firq_option==true`) and more than one register bank is present (`RGF_NUM_BANKS > 1`), software cannot set the `AUX_IRQ_CTRL.U` bit to 0; that is, the core does not support the option to save interrupt context to the kernel stack when a regular interrupt occurs in the user mode.

Saving and restoring of BLINK (R31) can be controlled independently of the general purpose pairs. Saving and restoring of the three loop-related registers LP_COUNT, LP_START and LP_END can be controlled as a group.

6.3.4.3.2 Exceptions During Interrupt Entry and Exit

Memory operations can take place during the automated saving and restoration of CPU context during interrupt entry and exit sequences. The interrupt entry and exit sequences are also referred to as prolog and epilog sequences, respectively. These memory operations can themselves raise memory-related exceptions, depending on the configuration. It is therefore possible to enter an exception handler as a result of attempting to automatically save CPU context prior to interrupt entry. It is also possible to enter an exception handler while restoring CPU context during the execution of an RTIE instruction.

If SecureShield 2+2 mode is configured, the software exception handler is responsible for restoring the processor to a state to continue execution.

The ARCV2 architecture guarantees that the interrupt entry or exit sequence can be replayed correctly.

The interrupt entry and exit pseudo code, describe the automatic save and restore of core registers. It is important to note that this code is not meant to imply the order in time of saving and restoring registers; only the layout in memory after a sequence successfully completes.

If an exception occurs midway through an prolog or epilog sequence, the user can make no assumption about which transfers to memory are complete as the stack area being written is considered to be unallocated until the sequence has committed or a secure to normal mode switch is complete (if SecureShield 2+2 mode is configured), at which point the stack pointer is updated.

Following a premature termination of a prolog or epilog sequence, after the exception is resolved and the exception handler executes the RTIE instruction, the failing prolog or epilog sequence is retried from the beginning.

The `AUX_IRQ_CTRL.U` bit, together with `STATUS32.U` and `STATUS32.S`, determines whether the user-mode stack pointer or the kernel-mode stack pointer is used during interrupt entry and exit sequences to

save and restore CPU context. All memory operations initiated within interrupt entry or exit sequences, are performed with kernel privilege if using the kernel-mode stack. Likewise, the memory operations are performed with user privilege if using the user-mode stack.

You should ensure that stack pointers are 32-bit aligned. The ARCV2 application binary interface (ABI) defines the pointers to be 32-bit aligned and ARCV2 implementations take advantage of this interface to minimize interrupt service latency. During execution of the interrupt prolog or epilog sequence, if the stack pointer is not 32-bit aligned, a misaligned access exception is always raised irrespective of the STATUS32.AD bit.

6.3.4.3.3 Regular Interrupt Entry

The following pseudo-code explains the sequence of a regular interrupt entry. To understand the pseudo code, refer to the following notation:

- P is the processor's current operating priority defined by the index of the least significant bit set (with value of 1) in AUX_IRQ_ACT.ACTIVE. P is 16 if no interrupts are active.
- Q is the index of the least significant bit set (with a value of 1) in AUX_IRQ_ACT.ACTIVE when bit P is cleared. Q is 16 if P is the only bit set to 1 in AUX_IRQ_ACT.ACTIVE.

Example 6-2 PI is the highest priority pending and enabled interrupt, and its priority is W. **Regular Interrupt Entry**

```
EnterInterrupt(PI, W) {
    // Assert ((W < P) && (STATUS32.IE) && (W ? STATUS32.E)
    //&& (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
    // indicates the prolog sequence in_int_prologue = 1;
    if (!AUX_IRQ_CTRL.U && STATUS32.U) {
        // early switch to kernel SP if in user thread
        AEX R28, [AUX_USER_SP]
        // speculatively switch to kernel mode
        STATUS32.U = 0;
    }
    ILINK = PC;
    ERET = PC;
    Push(STATUS32);
    Push(PC);
    if (AUX_IRQ_CTRL.LP && (JLI_OPTION || CD2_OPTION)) {
        if (JLI_OPTION == 1) {
            Push(JLI_BASE);
        }
        if (CD2_OPTION == 1) {
            Push(LDI_BASE);
            Push(EI_BASE);
        }
    }
    if (AUX_IRQ_CTRL.L) {
        Push(LP_COUNT);
        Push(LP_START);
        Push(LP_END);
    }
    if (RGF_NUM_REGS == 32) {
```

```

    if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16)) {
        Push(BLINK);
    }
    for (i = (2 * AUX_IRQ_CTRL.NR); i > 0; i -= 2) {
        Push(Ri-1);
        Push(Ri-2);
    }
}
// Else if RGF_NUM_REGS == 16
else {
    if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 8)) {
        Push(BLINK);
    }
    if (AUX_IRQ_CTRL.NR >= 8) { Push(R31); Push(R30); }
    if (AUX_IRQ_CTRL.NR >= 7) { Push(R29); Push(R28); }
    if (AUX_IRQ_CTRL.NR >= 6) { Push(R27); Push(R26); }
    if (AUX_IRQ_CTRL.NR >= 5) { Push(R15); Push(R14); }
    if (AUX_IRQ_CTRL.NR >= 4) { Push(R13); Push(R12); }
    if (AUX_IRQ_CTRL.NR >= 3) { Push(R11); Push(R10); }
    if (AUX_IRQ_CTRL.NR >= 2) { Push(R3); Push(R2); }
    if (AUX_IRQ_CTRL.NR >= 1) { Push(R1); Push(R0); }
}
if (AUX_IRQ_CTRL.U && STATUS32.U) {
    // late switch to kernel SP if in user thread
    AEX R28, [AUX_USER_SP];
}
if (AUX_IRQ_ACT.ACTIVE == 0) {
    AUX_IRQ_ACT.U = STATUS32.U;
}

AUX_IRQ_ACT.ACTIVE[W] = 1;
STATUS32.{Z = U; U = 0; L = 1; ES = 0; DZ = 0; DE = 0; }
// Out of interrupt prolog in_int_prologue = 1;
AcknowledgeInterrupt(PI);
Jump(InterruptVector(PI));
}

```

**Note**

The code in this example does not imply the order in time of memory operations; it only implies the layout in the memory. For more information, see [Exceptions During Interrupt Entry and Exit](#).

6.3.4.3.4 Regular Interrupt Entry When SecureShield 2+2 Mode is Configured

When SecureShield 2+2 Mode is configured, the pseudo-code does not modify the ERET register, and the order in which registers are pushed is different.

In SecureShield 2+2 Mode, ERET is used only for exception returns, and an interrupt return address is stored on the stack. SecureShield 2+2 Mode has to allow for both exceptions and secure/normal mode switching in the middle of the interrupt-entry sequence. In that case, it is up to the software to detect whether the exception occurred before or after mode switch and take the corresponding actions.

The order in which registers are pushed is different in SecureShield 2+2 Mode. For example, miscellaneous registers are pushed much later in SecureShield 2+2 Mode. The miscellaneous registers are pushed later in SecureShield 2+2 Mode because they are duplicated in secure and normal modes and both can be configured to be pushed to the stack, one to the secure stack and one to the normal stack. This allows software to assume those registers are in the same stack location even if a secure/normal mode switch occurs during IRQ entry.

Example 6-3 Regular Interrupt Entry When SecureShield 2+2 Mode is Configured

To understand the pseudo-code in this section, refer to the following notation:

- P is the processor's current operating priority defined by the index of the least significant bit set (with value of 1) in AUX_IRQ_ACT.ACTIVE. P is 16 if no interrupts are active.
- Q is the index of the least-significant bit set (with a value of 1) in AUX_IRQ_ACT.ACTIVE when bit P is cleared. Q is 16 if P is the only bit set to 1 in AUX_IRQ_ACT.ACTIVE.
- PI is the highest priority pending and enabled interrupt, and its priority is W.

Definitions of the functions used are provided at the end of the example.

```
/* When -sec_modes_option == true */
EnterInterrupt(PI, W) {
    /* Assert ((W < P) && (STATUS32.IE) && (W <= STATUS32.E)
    && (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
    /* indicates the prolog sequence in_int_prologue = 1 */
    ILINK = PC

    Check_vt(PI, AUX_IRQ_PRIORITY[PI].S)
    U_saved = STATUS32.U
    STATUS32_saved = STATUS32
    Set_uop_sid(0xff)

    /* switch to kernel stack if current U content needs to be pushed to K stack*/
    Early_switch_to_kernel = !AUX_IRQ_CTRL.U && STATUS32.U
    if (Early_switch_to_kernel == 1) {
        /* early switch to kernel SP if in user thread */
        Switch_sp(STATUS32.S, 1, STATUS32.S, 0)
        /* speculatively switch to kernel mode for following uops */
        STATUS32.U = 0
    }

    Push_STATUS_PC()
    Push_GPR()
    Switch_prologue_mode()
    Push_MISC()

    /* late switch to kernel if not switched already */
    if (Early_switch_to_kernel == 0 && STATUS32.U == 1) {
        Switch_sp(AUX_IRQ_PRIORITY(PI).S, 1, AUX_IRQ_PRIORITY(PI).S, 0)
    }

    if (AUX_IRQ_ACT.ACTIVE == 0) {
        AUX_IRQ_ACT.U = U_saved
    }
}
```

```

}
AUX_IRQ_ACT.ACTIVE[W] = 1
STATUS32.{Z = U_saved; U = 0; L = 1; ES = 0; DZ = 0; DE = 0; }

/* Out of interrupt prolog */
Check_handler_target(PI, AUX_IRQ_PRIORITY[PI].S)
int_prologue = 0
AcknowledgeInterrupt(PI)
if (AUX_IRQ_PRIORITY(PI).S) {
    Jump(INT_VECT_BASE_S(PI))
}
else {
    Jump(INT_VECT_BASE(PI))
}
/* UOP end, Processor start executing in mode of the fetching PC */
Set_mode_SID_by_MPU()
}

```

Definitions of Functions Used in the Code

```

Check_vt(PI, s){
    /* s is the variable indicate current mode, 1: S, 0: N */
    PC = (s ? INT_VECT_BASE_S : INT_VECT_BASE) + PI*4
    if (MPU_lookup(PC).overlap) {
        exception(EV_MachineCheck, ECR=0x030601)
    }
    if (PC in N mode and s == 1){
        exception(EV_MachineCheck, ECR = 0x031000);
    }
    if (PC in S mode and s == 0){
        exception(EV_ProtV, ECR = 0x061000);
    }
    if (MPU_lookup(PC).sid has multi-bits set){
        exception(EV_ProtV, ECR = 0x060044);
    }
}

Set_uop_sid(sid) {
    ifetch.sid = sid
    /*set the instruction fetch sid of following UOP as argument sid.*/
}

Switch_sp(ps, pu, s, u) {
    /* previous secure, previous user, current secure, current user */
    if (ps == s && pu == u) {
        return
    }
    Save_sp(ps, pu)
    Load_sp(s, u)
}

Save_sp(s, u) { /* secure, user */
    switch (s, u){
        case 0,0: AUX_KERNEL_SP = SP

```

```

        case 0,1: AUX_USER_SP    = SP
        case 1,0: AUX_SEC_K_SP   = SP
        case 1,1: AUX_SEC_U_SP   = SP
    }
}

Load_sp(s, u) { /* secure, user */
    switch (s, u) {
        case 0,0: SP = AUX_KERNEL_SP
        case 0,1: SP = AUX_USER_SP
        case 1,0: SP = AUX_SEC_K_SP
        case 1,1: SP = AUX_SEC_U_SP
    }
}

Push_STATUS_PC() {
    Push(STATUS32_saved)
    /* all bits are pushed if current mode is S, otherwise only IRM is pushed */
    if (STATUS32.S == 1) {
        SEC_STAT_PUSH = SEC_STAT
    }
    else {
        SEC_STAT_PUSH = 0
        SEC_STAT_PUSH.IRM = SEC_STAT.IRM
    }
    Push(SEC_STAT_PUSH)
    Push(ILINK)
}

Push_GPR() {
    /* clear GPR for S to N IRQ prolog */
    GPR_clear = (STATUS32.S == 1 && AUX_IRQ_PRIORITY[PI].S == 0)

    if (HAS_ACCUM && GPR_clear) {
        Push(ACC_L); ACC_L = 0
        Push(ACC_H); ACC_H = 0
    }
    if (RGF_NUM_REGS == 32) {
        if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16) && !GPR_clear) {
            Push(BLINK)
        }
        GPR_NUM = GPR_clear ? 32 : (2 * AUX_IRQ_CTRL.NR)
        for (i = GPR_NUM - 1; i >= 0; i -= 1) {
            Push(Ri)
            /* SP not clear, stack in use */
            if (i != 28 && GPR_clear) { Ri = 0 }
        }
    }

    // else if RGF_NUM_REGS == 16
    else {
        if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 8) && !GPR_clear) {

```



```

        Push(BLINK)
    }

    if (AUX_IRQ_CTRL.NR >= 8 || GPR_clear) {
        Push(R31); if (GPR_clear) { R31 = 0 }
        Push(R30); if (GPR_clear) { R30 = 0 }
    }
    if (AUX_IRQ_CTRL.NR >= 7 || GPR_clear) {
        Push(R29); if (GPR_clear) { R29 = 0 }
        /* SP not clear, stack in use*/
        Push(R28)
    }
    if (AUX_IRQ_CTRL.NR >= 6 || GPR_clear) {
        Push(R27); if (GPR_clear) { R27 = 0 }
        Push(R26); if (GPR_clear) { R26 = 0 }
    }
    if (AUX_IRQ_CTRL.NR >= 5 || GPR_clear) {
        Push(R15); if (GPR_clear) { R15 = 0 }
        Push(R14); if (GPR_clear) { R14 = 0 }
    }
    if (AUX_IRQ_CTRL.NR >= 4 || GPR_clear) {
        Push(R13); if (GPR_clear) { R13 = 0 }
        Push(R12); if (GPR_clear) { R12 = 0 }
    }
    if (AUX_IRQ_CTRL.NR >= 3 || GPR_clear) {
        Push(R11); if (GPR_clear) { R11 = 0 }
        Push(R10); if (GPR_clear) { R10 = 0 }
    }
    if (AUX_IRQ_CTRL.NR >= 2 || GPR_clear) {
        Push( R3); if (GPR_clear) { R3  = 0 }
        Push( R2); if (GPR_clear) { R2  = 0 }
    }
    if (AUX_IRQ_CTRL.NR >= 1 || GPR_clear) {
        Push( R1); if (GPR_clear) { R1  = 0 }
        Push( R0); if (GPR_clear) { R0  = 0 }
    }
}

}

Push_MISC() {
    if (AUX_IRQ_CTRL.LP && (JLI_OPTION || CD2_OPTION)) {
        if (JLI_OPTION == 1) {
            Push(JLI_BASE)
        }
        if (CD2_OPTION == 1) {
            Push(LDI_BASE)
            Push(EI_BASE)
        }
    }
}

if (AUX_IRQ_CTRL.L) {
    Push(LP_COUNT)
    Push(LP_START)
    Push(LP_END)
}

```

```

    }
}

Switch_prologue_mode() {
    /* NSRT/U bits are updated on S to N IRQ prologue only */
    if (STATUS32.S == 1 && AUX_IRQ_PRIORITY[PI].S == 0) {
        SEC_STAT.NSRT = 1
        SEC_STAT.NSRU = U_saved
    }
    SEC_STAT.IRM = STATUS32.S

    if (Early_switch_to_kernel == 1) {
        /* save early switch kernel SP */
        Switch_sp(STATUS32.S, 0, AUX_IRQ_PRIORITY(PI).S, 0)
    } else {
        Switch_sp(STATUS32.S, STATUS32.U, AUX_IRQ_PRIORITY(PI).S, STATUS32.U)
    }
    Switch_duplicated_regs(AUX_IRQ_PRIORITY(PI).S)
    /* S/N mode switch */
    STATUS32.S = AUX_IRQ_PRIORITY(PI).S
}

Switch_duplicated_regs(mode) {
    /* mode argument indicates the mode to switch to (mode == 0: N; mode == 1: S). Those
    operations are done automatically when current instruction/UOP change N/S mode, no UOP
    is issued as a result of duplicated register switch */
    LP_END(!mode) = LP_END          /* save ! mode version */
    LP_END = LP_END(mode)          /* load mode version */
    LP_START(!mode) = LP_START      /* save ! mode version */
    LP_START = LP_START(mode)      /* load mode version */
    LP_COUNT(!mode) = LP_COUNT      /* save ! mode version */
    LP_COUNT = LP_COUNT(mode)      /* load mode version */
    EI_BASE(!mode) = EI_BASE        /* save ! mode version */
    EI_BASE = EI_BASE(mode)        /* load mode version */
    LDI_BASE(!mode) = LDI_BASE      /* save ! mode version */
    LDI_BASE = LDI_BASE(mode)      /* load mode version */
    JLI_BASE(!mode) = JLI_BASE      /* save ! mode version */
    JLI_BASE = JLI_BASE(mode)      /* load mode version */
    BTA(!mode) = BTA               /* save ! mode version */
    BTA = BTA(mode)               /* load mode version */
}

Check_handler_target(PI, s) {
    /* s is the variable indicate current mode, 1: S, 0: N */
    ENTRY_ADDR = (s ? INT_VECT_BASE_S : INT_VECT_BASE) + PI*4
    TARGET_ADDR = [ENTRY_ADDR]
    if (MPU_lookup(Target_ADDR).overlap) {
        exception(EV_MachineCheck, ECR=0x030601)
    }
    if (TARGET_ADDR in N mode and s == 1) {
        exception(EV_MachineCheck, ECR = 0x031200)
    }
    if (TARGET_ADDR in S mode and s == 0) {

```

```

        exception(EV_ProtV , ECR = 0x061100)
    }
}

Set_mode_SID_by_MPU() {
    /*set the mode and sid look up from fetch MPU */
    ifetch.SID = MPU_lookup(PC).SID
    STATUS32.S = MPU_lookup(PC).S
}

```

Example 6-4 Regular Interrupt Entry

```

EnterInterrupt(PI, W){
    // Assert ((W < P) && (STATUS32.IE) && (W ? STATUS32.E)
    //&& (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
    // indicates the prolog sequence in_int_prologue = 1;
    if (!AUX_IRQ_CTRL.U && STATUS32.U) {
        // early switch to kernel SP if in user thread
        AEX R28, [AUX_USER_SP]
        // speculatively switch to kernel mode
        STATUS32.U = 0;
    }
    ILINK = PC;
    ERET = PC;
    Push(STATUS32);
    Push(PC);
    if (AUX_IRQ_CTRL.LP && (JLI_OPTION || CD2_OPTION)) {
        if (JLI_OPTION == 1) {
            Push(JLI_BASE);
        }
        if (CD2_OPTION == 1) {
            Push(LDI_BASE);
            Push(EI_BASE);
        }
    }
    if (AUX_IRQ_CTRL.L) {
        Push(LP_COUNT);
        Push(LP_START);
        Push(LP_END);
    }
    if (RGF_NUM_REGS == 32) {
        if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16)) {
            Push(BLINK);
        }
        for (i = (2 * AUX_IRQ_CTRL.NR); i > 0; i -= 2) {
            Push(Ri-1);
            Push(Ri-2);
        }
    }
    // Else if RGF_NUM_REGS == 16
    else {
        if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 8)) {

```

```

        Push (BLINK) ;
    }
    if (AUX_IRQ_CTRL.NR >= 8) { Push(R31); Push(R30); }
    if (AUX_IRQ_CTRL.NR >= 7) { Push(R29); Push(R28); }
    if (AUX_IRQ_CTRL.NR >= 6) { Push(R27); Push(R26); }
    if (AUX_IRQ_CTRL.NR >= 5) { Push(R15); Push(R14); }
    if (AUX_IRQ_CTRL.NR >= 4) { Push(R13); Push(R12); }
    if (AUX_IRQ_CTRL.NR >= 3) { Push(R11); Push(R10); }
    if (AUX_IRQ_CTRL.NR >= 2) { Push( R3); Push( R2); }
    if (AUX_IRQ_CTRL.NR >= 1) { Push( R1); Push( R0); }
}
if (AUX_IRQ_CTRL.U && STATUS32.U) {
    // late switch to kernel SP if in user thread
    AEX R28, [AUX_USER_SP];
}
if (AUX_IRQ_ACT.ACTIVE == 0) {
    AUX_IRQ_ACT.U = STATUS32.U;
}

AUX_IRQ_ACT.ACTIVE[W] = 1;
STATUS32.{Z = U; U = 0; L = 1; ES = 0; DZ = 0; DE = 0; }
// Out of interrupt prolog in_int_prologue = 1;
AcknowledgeInterrupt(PI);
Jump(InterruptVector(PI));
■ }

```

6.3.4.4 Returning from Regular Interrupts

When the interrupt routine is entered, the corresponding priority level bit *i* in `AUX_IRQ_ACT` is set, thus preventing other interrupts at the same level or below from preempting the taken interrupt.

The return from interrupt or exception instruction, `RTIE`, allows exit from interrupt and exception handlers, and also allows the processor to switch from kernel mode to user mode.

The `RTIE` instruction is available only in kernel mode. Using this instruction in the user mode raises a Privilege Violation, Kernel Only Access exception.

The return from an interrupt is determined by the `SEC_STAT.IRM` bit. If the mode that the `RTIE` instruction returns the core to different from the mode that the `SEC_STAT.IRM` bit permits, an `EV_EV_PrivilegeV` exception is raised.

Use the `RTIE` instruction to exit an interrupt or exception handlers. The `RTIE` instruction restores previous context including the program counter, status register and, optionally, selected core registers depending on whether the return is from an exception, a fast or regular interrupt, and to what machine operating level the processor is returning.

Bits in the `STATUS32` ([Status Register, STATUS32](#)), `AUX_IRQ_ACT` ([Active Interrupts Register, AUX_IRQ_ACT](#)) and `IRQ_PRIORITY_PENDING` ([Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING](#)) registers are provided to allow the `RTIE` instruction to determine what pre-interrupt or exception machine state to restore and from where to reload the state. When returning from a

regular interrupt, selected core and auxiliary registers may also be restored from the stack depending on bits in the AUX_IRQ_CTRL ([Interrupt Context Saving Control Register, AUX_IRQ_CTRL](#) register).

On return from a fast interrupt the processor restores the STATUS32 register including the RB field. When more than one bank of core registers is configured, the processor implicitly switches to the register bank defined by the RB field in the restored STATUS32 register. A detailed discussion of the actions taken on interrupt and exception entry and exit is available in sections:

- [Fast Interrupt Entry](#)
- [Fast Interrupt Exit](#)
- [Regular Interrupt Entry](#)
- [Regular Interrupt Exit](#)
- [Exception Entry](#)
- [Exception Exit](#)
- [Exceptions and Delay Slots](#)

When the processor is returning from an interrupt and another enabled interrupt is pending, the processor avoids restoring the registers on interrupt exit by comparing the priority level of the exiting interrupt to the highest pending priority interrupt.



Note

Tail chaining of interrupt handlers for pending interrupts (starting a second ISR immediately following a previous ISR without pop/push of any registers) is supported only when the handler mode for both interrupts is the same (a secure interrupt is followed by a secure interrupt or a normal interrupt is followed by a normal interrupt). Where a mode change is required (secure to normal or normal to secure) the normal interrupt return protocol is followed prior to taking the second interrupt.

See the following example to understand how the processor can save redundant restore and save instructions during exiting interrupts.

Assume, for example, that the processor is servicing a priority level P4 interrupt and another interrupt P5 with a priority, W, is pending.

When exiting interrupt P4, the RTIE instruction has to restore the registers from the stack and again save the same registers to the stack when entering the P5 interrupt. This restore and saving of the same registers is a redundant action from the processor. To remove this redundancy when exiting the P4 interrupt, the processor verifies if interrupts are still enabled and if P5 is the highest pending interrupt and within the priority threshold ($W \leq \text{STATUS32.E}$); if P5 is serviceable, the processor does not restore the stack.

Similarly, if the processor is exiting a fast interrupt and another fast interrupt is pending and interrupts are enabled, the processor jumps to the new interrupt vector instead of restoring the PC and STATUS32 registers.

6.3.4.4.1 Regular Interrupt Exit

The following pseudo-code explains the sequence of a regular interrupt exit:

The following example explains the sequence of a regular interrupt entry. To understand the pseudo code, refer to the following notation:

- P is the processor's current operating priority defined by the index of the least significant bit set (with value of 1) in AUX_IRQ_ACT.ACTIVE. P is 16 if no interrupts are active.
- Q is the index of the least significant bit set (with a value of 1) in AUX_IRQ_ACT.ACTIVE when bit P is cleared. Q is 16 if P is the only bit set to 1 in AUX_IRQ_ACT.ACTIVE.
- PI is the highest priority pending and enabled interrupt, and its priority is W.

Definitions of the functions used are provided at the end of the example.

Example 6-5 Regular Interrupt Exit

```
ReturnFromInterrupt (PI, W){
    //Assert(((W >= P) || (W < STATUS32.E)) && ((P > 0) || (FIRQ_OPTION == 0)) &&
    //(STATUS32.AE == 0))
    // clear the currently-active interrupt
    AUX_IRQ_ACT.ACTIVE[P] = 0;
    // if pending interrupts, avoid pop/push
    if (((W > 0) || (FIRQ_OPTION == 0)) && ((W < Q) && (W ? STATUS32.E))) {
        // set new interrupt level
        AUX_IRQ_ACT.ACTIVE[W] = 1;
        STATUS32.L = 1;
        STATUS32.Z = U;
        // acknowledge interrupt
        AcknowledgeInterrupt(PI);
        // jump to interrupt vector
        Jump(InterruptVector(PI));
    }
    // else pop context before interrupt exit
    else {
        // if returning to user thread and user stack stores user context
        if ((AUX_IRQ_ACT.ACTIVE == 0) && AUX_IRQ_ACT.U && AUX_IRQ_CTRL.U) {
            AEX R28, [AUX_USER_SP];
            STATUS32.U = 1;
        }
        // If the register bank has 32 entry registers files
        if (RGF_NUM_REGS == 32) {
            // pop AUX_IRQ_CTRL.NR register pairs
            for (i = 0; i < (2 * AUX_IRQ_CTRL.NR); i += 2){
                if (i == 28) {
                    /* not restore SP */
                    Pop(tmp);
                    Pop(Ri+1);
                }
                else{
                    Pop(Ri);
                    Pop(Ri+1);
                }
            }
            // if BLINK should be restored separately
            if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16)){
                Pop(BLINK);
            }
        }
    }
}
```

```

    }

    // If the register bank has 16 entry register files
    else {
        if (AUX_IRQ_CTRL.NR >= 1) { Pop( R0); Pop( R1); }
        if (AUX_IRQ_CTRL.NR >= 2) { Pop( R2); Pop( R3); }
        if (AUX_IRQ_CTRL.NR >= 3) { Pop(R10); Pop(R11); }
        if (AUX_IRQ_CTRL.NR >= 4) { Pop(R12); Pop(R13); }
        if (AUX_IRQ_CTRL.NR >= 5) { Pop(R14); Pop(R15); }
        if (AUX_IRQ_CTRL.NR >= 6) { Pop(R26); Pop(R27); }
        if (AUX_IRQ_CTRL.NR >= 7) { Pop(R28); Pop(R29); }
        if (AUX_IRQ_CTRL.NR >= 8) { Pop(R30); Pop(R31); }
        if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 8)) {
            Pop(BLINK);
        }
    }
}
// restore ZOL state
if (AUX_IRQ_CTRL.L) {
    Pop(LP_END);
    Pop(LP_START);
    Pop(LP_COUNT);
}
// restore Code Density State
if (AUX_IRQ_CTRL.LP && (CD2_OPTION || JLI_OPTION)) {
    if (CD2_OPTION == 1) {
        Pop(EI_BASE);
        Pop(LDI_BASE);
    }
    if (JLI_OPTION == 1) {
        Pop(JLI_BASE);
    }
}
// pop PC
Pop(PC);
// pop STATUS32
Pop(STATUS32);
AUX_IRQ_ACT.U = 0;

// if returning to user thread and kernel stack stores user context
if ( (AUX_IRQ_ACT.ACTIVE == 0) && AUX_IRQ_ACT.U && !AUX_IRQ_CTRL.U) {
    AEX R28, [AUX_USER_SP];
    AUX_IRQ_ACT.U = 0;
}
// jump back to outer context
Jump(PC);

```

Definitions of Functions Used in the Code

```

Switch_duplicated_regs(mode) {
/* mode argument indicates the mode to switch to (mode == 0: N; mode == 1: S). Those operations are done automatically when current instruction/UOP change N/S mode, no UOP is issued as a result of duplicated register switch */
    LP_END(!mode) = LP_END          /* save ! mode version */

```

```

LP_END = LP_END(mode)          /* load mode version */
LP_START(!mode) = LP_START     /* save ! mode version */
LP_START = LP_START(mode)      /* load mode version */
LP_COUNT(!mode) = LP_COUNT     /* save ! mode version */
LP_COUNT = LP_COUNT(mode)      /* load mode version */
EI_BASE(!mode) = EI_BASE       /* save ! mode version */
EI_BASE= EI_BASE(mode)         /* load mode version */
LDI_BASE(!mode) = LDI_BASE     /* save ! mode version */
LDI_BASE= LDI_BASE(mode)       /* load mode version */
JLI_BASE(!mode) = JLI_BASE     /* save ! mode version */
JLI_BASE= JLI_BASE(mode)       /* load mode version */
BTA(!mode) = BTA               /* save ! mode version */
BTA= BTA(mode)                 /* load mode version */
}

```

Example 6-6 Regular Interrupt Exit When SecureShield 2+2 Mode is Configured

```

ReturnFromInterrupt (PI, W){
    /* Assert(((W >= P) || (W < STATUS32.E))) && ((P > 0) || (FIRQ_OPTION == 0))
       && (STATUS32.AE == 0)) */

    Check_Mismatch()

    /* clear the currently-active interrupt */
    AUX_IRQ_ACT.ACTIVE[P] = 0

    /* chaining only if PI IRQ mode is the same as current mode */
    if (((W > 0) || (FIRQ_OPTION == 0)) && ((W < Q) && (W <= STATUS32.E))
        && (STATUS32.S == AUX_IRQ_PRIORITY[PI].S)) {
        in_int_prologue = 1
        Check_vt(PI, AUX_IRQ_PRIORITY[PI].S)
        Check_handler_target(PI, aux_irq_priority[PI].S)

        /* set new interrupt level */
        AUX_IRQ_ACT.ACTIVE[W] = 1
        STATUS32.L = 1
        STATUS32.Z = STATUS32.U

        /* acknowledge interrupt */
        AcknowledgeInterrupt(PI)
        int_prologue = 0

        /* jump to interrupt vector */
        if (AUX_IRQ_PRIORITY[PI].S)
            Jump(INT_VECT_BASE_S(PI))
        else
            Jump(INT_VECT_BASE(PI))
        /* UOP end, Processor start executing in mode of the fetching PC */
        Set_mode_SID_by_MPU()
    }
    /* else pop context before interrupt exit */
    else {

```



```

Set_uop_sid(0xff)
/* Switch to User stack if context is saved to U stack */
Early_switch_to_user = AUX_IRQ_ACT.ACTIVE == 0 && AUX_IRQ_ACT.U && AUX_IRQ_CTRL.U

if ( Early_switch_to_user == 1) {
    /* early switch to user stack */
    Switch_sp(STATUS32.S, 0, STATUS32.S, 1)
    STATUS32.U = 1
}
IRM_saved = SEC_STAT.IRM
S_saved = STATUS32.S

Pop_MISC()
Switch_epilogue_mode(IRM_saved)
Pop_GPR(IRM_saved, S_saved)
Pop_STATUS_PC(IRM_saved)

/* late switch to user */
if (AUX_IRQ_ACT.ACTIVE == 0 && AUX_IRQ_ACT.U && !AUX_IRQ_CTRL.U) {
    Switch_sp(IRM_saved, 0, IRM_saved, 1)
}
/* jump back to outer context */
Check_jump_target(IRM_saved, ILINK)
Jump(ILINK)
/* UOP end, Processor start executing in mode of the fetching PC */
Set_mode_SID_by_MPU()
}

```

Definitions of Functions Used in the Code

```

Check_Mismatch() {
    /* only check on N to S IRQ return */
    if (!(STATUS32.S == 0 && SEC_STAT.IRM == 1)) {
        return
    }
    if (SEC_STAT.NSRT == 0) {
        Raise_wrong_return_type_exception()
    }
    TMP = AUX_IRQ_ACT.ACTIVE
    TMP[P] = 0

    if (TMP == 0 && AUX_IRQ_ACT.U && SEC_STAT.NSRU == 0) {
        /* u bit mismatch, wrong U/K stack detected */
        Raise_u_mismatch_exception
    }
}

Raise_wrong_return_type_exception() {
    exception(EV_PrivilegeV , ECR = 0x071001)
}

Raise_u_mismatch_exception() {
    exception(EV_PrivilegeV , ECR = 0x071002)
}

```

```

}

Check_vt(PI, s){
    /* s is the variable indicate current mode, 1: S, 0: N */
    PC = (s ? INT_VECT_BASE_S : INT_VECT_BASE) + PI*4
    if (MPU_lookup(PC).overlap) {
        exception(EV_MachineCheck, ECR=0x030601)
    }
    if (PC in N mode and s == 1){
        exception(EV_MachineCheck, ECR = 0x031000);
    }
    if (PC in S mode and s == 0){
        exception(EV_ProtV, ECR = 0x061000);
    }
    if (MPU_lkup(PC).sid has multi-bits set){
        exception(EV_ProtV, ECR = 0x060044);
    }
}

Pop_MISC(){
    /* restore ZOL state */
    if (AUX_IRQ_CTRL.L) {
        Pop(LP_END)
        Pop(LP_START)
        Pop(LP_COUNT)
    }
    /* restore Code Density State */
    if (AUX_IRQ_CTRL.LP && (CD2_OPTION || JLI_OPTION)) {
        if (CD2_OPTION == 1) {
            Pop(EI_BASE)
            Pop(LDI_BASE)
        }
        if (JLI_OPTION == 1) {
            Pop(JLI_BASE)
        }
    }
}

Switch_epilogue_mode(IRM_saved){
    if (Early_switch_to_User == 1) {
        Switch_sp(STATUS32.S, 1, IRM_saved, 1)
    } else {
        Switch_sp(STATUS32.S, 0, IRM_saved, 0)
    }
    Switch_duplicated_regs(IRM_saved)
    /* S/N mode switch */
    STATUS32.S = IRM_saved
}

Switch_duplicated_regs(mode){
    /* mode argument indicates the mode to switch to (mode == 0: N; mode == 1: S). Those
    operations are done automatically when current instruction/UOP change N/S mode, no UOP
    is issued as a result of duplicated register switch */

```

```

    LP_END(!mode) = LP_END          /* save ! mode version */
    LP_END = LP_END(mode)           /* load mode version */
    LP_START(!mode) = LP_START      /* save ! mode version */
    LP_START = LP_START(mode)       /* load mode version */
    LP_COUNT(!mode) = LP_COUNT      /* save ! mode version */
    LP_COUNT = LP_COUNT(mode)       /* load mode version */
    EI_BASE(!mode) = EI_BASE        /* save ! mode version */
    EI_BASE= EI_BASE(mode)          /* load mode version */
    LDI_BASE(!mode) = LDI_BASE      /* save ! mode version */
    LDI_BASE= LDI_BASE(mode)        /* load mode version */
    JLI_BASE(!mode) = JLI_BASE      /* save ! mode version */
    JLI_BASE= JLI_BASE(mode)        /* load mode version */
    BTA(!mode) = BTA                /* save ! mode version */
    BTA= BTA(mode)                  /* load mode version */
}

Switch_sp(ps, pu, s, u) {
    /* previous secure, previous user, current secure, current user */
    if (ps == s && pu == u) {
        return
    }
    Save_sp(ps, pu)
    Load_sp(s, u)
}

Pop_GPR(IRM_saved, S_saved){
    N2S = S_saved == 0 && IRM_saved == 1
    /* if the register bank has 32 entry registers files */
    if (RGF_NUM_REGS == 32) {
        /* Pop AUX_IRQ_CTRL.NR register pairs or all Regs */
        GPR_NUM = N2S ? 32 : (2 * AUX_IRQ_CTRL.NR)
        for (i = 0; i < GPR_NUM; i += 1){
            if (i == 28) { /* not restore SP */
                Pop(tmp);
                Pop(Ri+1);
            }
            else {
                Pop(Ri);
                Pop(Ri+1);
            }
        }
    }
}

if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16) && !N2S) {
    Pop(BLINK)
}

/* if the register bank has 16 entry register files */
else {
    if (AUX_IRQ_CTRL.NR >= 1 || N2S) { Pop( R0); Pop( R1) }
    if (AUX_IRQ_CTRL.NR >= 2 || N2S) { Pop( R2); Pop( R3) }
    if (AUX_IRQ_CTRL.NR >= 3 || N2S) { Pop(R10); Pop(R11) }
    if (AUX_IRQ_CTRL.NR >= 4 || N2S) { Pop(R12); Pop(R13) }
    if (AUX_IRQ_CTRL.NR >= 5 || N2S) { Pop(R14); Pop(R15) }
}

```

```

        if (AUX_IRQ_CTRL.NR >= 6 || N2S) { Pop(R26); Pop(R27) }
        if (AUX_IRQ_CTRL.NR >= 7 || N2S) {
            Pop(tmp) /* not restore SP */
            Pop(R29)
        }
        if (AUX_IRQ_CTRL.NR >= 8 || N2S) { Pop(R30); Pop(R31) }
        if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 8) && !N2S) {
            Pop(BLINK)
        }
    }
    if (HAS_ACCUM && N2S) {
        Pop(ACC_H)
        Pop(ACC_L)
    }
}

Pop_STATUS_PC(IRM_saved) {
    /* pop PC */
    Pop(ILINK)

    Pop(SEC_STAT_POP)
    /* NSRT/U SSC etc are restored only if return to S */
    if (IRM_saved == 1) {
        SEC_STAT = SEC_STAT_POP
    } else {
        SEC_STAT.IRM = SEC_STAT_POP.IRM
    }
    Pop(STATUS32)
}

Check_handler_target(PI, s) {
    /*s is the variable indicate current mode, 1: S, 0: N*/
    ENTRY_ADDR = (s ? INT_VECT_BASE_S : INT_VECT_BASE) + PI*4
    TARGET_ADDR = [ENTRY_ADDR]
    if (MPU_lookup(Target_ADDR).overlap) {
        exception(EV_MachineCheck, ECR=0x030601)
    }
    if (TARGET_ADDR in N mode and s == 1) {
        exception(EV_MachineCheck , ECR = 0x031200)
    }
    if (TARGET_ADDR in S mode and s == 0) {
        exception(EV_ProtV , ECR = 0x061100)
    }
}

Check_jmp_target(s, addr) {
    if (MPU_lookup(addr).overlap) {
        exception(EV_MachineCheck, ECR=0x030601)
    }
    /* is the variable indicate current mode, 1: S, 0: N */
    if (addr in N mode and s == 1) {
        exception(EV_PrivilegeV, ECR = 0x071003)
    }
}

```

```

    if (addr in S mode and s == 0){
        exception(EV_PrivilegeV, ECR =0x071003)
    }
}

Set_uop_sid(sid) {
    ifetch.sid = sid
    /*set the instruction fetch sid of following UOP as argument sid.*/
}

Set_mode_SID_by_MPU() {
    /*set the mode and sid look up from fetch MPU */
    ifetch.SID = MPU_lookup(PC).SID
    STATUS32.S = MPU_lookup(PC).S
}

```

**Note**

The code in this example does not imply the order in time of memory operations; it only implies the layout in the memory. For more information, see [Exceptions During Interrupt Entry and Exit](#).

6.3.4.5 Interrupt Timing

Interrupts are held off when a compound instruction has a dependency on the following instruction or is waiting for immediate data from memory. This dependency occurs during a branch, jump, or simply when an instruction uses long immediate data. The time taken to service an interrupt encompasses a jump to the appropriate vector, and then a jump to the routine pointed to by that vector. The timings of interrupts, according to the configuration of the processor, are defined by each ARCV2 processor Databook.

The time taken to service an interrupt also depends on the following:

- Whether an I- Cache miss occurs when accessing the interrupt vector table or when fetching the instructions at the entry point of the service routine
- The number of registers saved to a software stack at the start of the interrupt service routine
- Whether an interrupt of the same or higher level is already being serviced
- Whether the interrupt is itself interrupted by a higher level interrupt or an exception

6.3.4.6 Determine Active Interrupts

The following pseudo-code demonstrates how to determine the active interrupts in a processor:

Figure 6-3 Determine Active Interrupts

```

scan_interrupt_vectors()
{
    CLRI R0 ;; disable interrupts before manipulating IRQ_SELECT
    for all interrupt numbers x from 0 to 255
    do {
        write x to the IRQ_SELECT register
        write x to the AUX_IRQ_HINT register
        read from the IRQ_STATUS register, and if the IRQ_STATUS.IP bit (31) is set to 1,
        vector x is an interrupt
    }
}

```

```

        Otherwise vector x is not an interrupt.
        If x < 16, x is an exception vector, otherwise x is beyond the defined interrupt
        region.
    }
    write 0 to the AUX_IRQ_HINT register to clear all software interrupt hints
    SETI R0 ;; re-enable interrupts
}

```

6.4 Exceptions

The processor is designed to allow exceptions to be taken and handled from user mode or kernel mode, and from interrupt service routines. However, an exception taken in an exception handler is a *double fault* condition, and raises a fatal Machine Check exception.

All interrupts and exceptions cause an immediate switch into the kernel mode. The Memory Management Unit (if present) is not disabled on entry to an interrupt or exception handler, and the process-ID (ASID) register is not altered. Interrupts are disabled on entry to an exception handler.

You can program the modes in which an exception is handled. By default, Reset and certain exception handlers are always handled in the secure kernel mode only (see [Table 6-6](#)). The Trap exception handler is always handled in the mode that triggered the exception. The other exception handlers can be programmed to be handled in the secure kernel mode or the mode they were triggered from.

Use the [AUX_SEC_EXCEPT](#) register to program the modes in which each exception is handled.

For double exception, if the first exception is a secure exception, the second exception will also be a secure exception. If the first exception is a normal exception, and if the second exception is a secure exception, the exception status registers do not need protection because the saved information is normal.

6.4.1 Exception Precision

The term **precise exception** refers to a synchronous exception event associated with a specific instruction. When a precise exception is taken, the exception handler knows precisely the point in the program at which the exception took place. If a precise exception can be restarted, all state changes from instructions that occurred before the exception point are completed, and none of the state changes from instructions that occur after the exception point are completed.

All ARCV2-based processors implement precise exceptions, and all instructions can be restarted up and until they are committed. Apart from instructions that retire post-commit, instructions can always be dismissed before completion and restarted later. On receipt of an exception, an operating system can do any of the following:

- Kill the process
- Send a signal to the process
- Intervene to remove the cause of the exception, and restart the process

A memory error exception may not be recoverable depending on the actual event that caused the memory error. For example:

- An instruction cache load that causes a bus error, and therefore a [Machine Check, Internal Instruction Memory Error](#), is precise because the address of the instruction is known at the time of the memory error.

- A load or store operation that causes a bus error, and therefore a [Bus Error on Data Access to External Data Memory](#) exception, may be imprecise if the processor is configured with a data cache that has a copy-back policy. In such cases, a copy-back operation may fail when writing a block from cache to an address that originated from an earlier load or store instruction. If this copy-back operation occurs, the program would have progressed beyond the originating load or store instruction, and therefore, the exception is imprecise and irrecoverable.

Imprecise exceptions are asynchronous events that may or may not be associated with a specific instruction. When an imprecise exception is raised and the processor is in exception mode, the first such delayed exception is taken after the processor has exited the Exception mode. Any second or subsequent imprecise exceptions are discarded.

**Note**

When an imprecise exception is raised and the processor is in the exception mode, the first such delayed exception is taken after the processor has exited the Exception mode. However, in this situation imprecise exceptions raised by the vector unit (EV_VecUnit exception or EV_MachineCheck exception with cause code 0x80) are not delayed, but are instead converted to an EV_MachineCheck double fault exception.

6.4.1.1 Non-Maskable Imprecise Exception

You can trigger a high priority non-maskable imprecise exception by asserting the external signal: `nmi`. Use the `-nmi_option` to configure this support. When the `nmi` signal is asserted, a non-maskable imprecise exception is raised with ECR 0x031400 cause code. You can use this exception in situations where non-correctable errors need immediate attention and situations where you want to restore the core from lockup status and read the debug information.

Related Topics:

- See [Exception Types and Priorities](#) for information on the non-maskable imprecise exception priority
- See [Machine Check, Non-Maskable Imprecise Exception, Double fault](#)

6.4.2 Exception Vectors and the Exception Cause Register

Vectors are fetched in instruction space and thus may be present in ICCM, Instruction Cache, or main memory accessed by instruction fetch logic. Every exception has the following associated information:

- Vector Name
- Vector Number
- Vector Offset
- Cause Code
- Parameter

6.4.2.1 Vector Name

The vector name is a symbolic equivalent to the vector number.

6.4.2.2 Vector Number

An 8-bit index into the table of interrupt or exception vectors that is unique to each distinct exception or interrupt supported by the system.

6.4.2.3 Vector Offset

The Vector Offset is used to determine the position of the appropriate interrupt or exception service routine for a given interrupt or exception. The vector offset is calculated as four times the vector number, and is an offset from the interrupt or exception vector base address.

The vector offsets are summarized in [Table 6-5](#).

Table 6-5 Interrupt/Exception Vectors

Name	Vector Offset	Vector Number	Exception Types
Reset	0x00	0x00	Exception
Memory Error	0x04	0x01	Exception
Instruction Error	0x08	0x02	Exception
EV_MachineCheck	0x0C	0x03	Exception
EV_TLBMissI	0x10	0x04	Exception
EV_TLBMissD	0x14	0x05	Exception
EV_ProtV	0x18	0x06	Exception
EV_PrivilegeV	0x1C	0x07	Exception
EV_SWI	0x20	0x08	Exception
EV_Trap	0x24	0x09	Exception
EV_Extension	0x28	0x0A	Exception
EV_DivZero	0x2C	0x0B	Exception
EV_DCError	0x30	0x0C	Exception
EV_Misaligned	0x34	0x0D	Exception
EV_VecUnit	0x38	0x0E	Vector unit exceptions
Unused	0x3C	0x0F	
IRQ16 - IRQ 255	0x40 - 0x3FC	0x10 - FF	Interrupts

The entries in an exception vector table contain addresses of interrupt and exception handlers, not jump instructions to those handlers. Although the processor always correctly jumps to a handler if an exception or an interrupt happens, execution of a jump instruction to the vector table has undefined behavior.

[Table 6-6](#) provides more information about the exception priorities and exception cause parameters.

If you need to explicitly pass control to an interrupt handler, use the `AUX_IRQ_HINT` register.

To pass control to a handler or perform a soft reset, do the following:

Example 6-7 Pass Control to an Handler or Perform a Soft Reset

1. Read the contents of a the vector table entry into a core register.
2. Execute an indirect jump on the core register.

The following code shows how a soft reset can be implemented:

```
LR %R0, [%INT_VECTOR_BASE]
LD %R1, [%R0]
J [%R1]
```

Example 6-8 Exception Vector Code

```
_reset:                ; entry point
.long _start           ; exception 0 offset 0x0    0
.long memory_error     ; exception 1 offset 0x4    4
.long instruction_error ; exception 2 offset 0x8    8
```

6.4.2.4 Cause Codes

Because multiple exceptions share each vector, this 8-bit number is used to identify the exact cause of an exception.

6.4.2.5 Parameters

This 8-bitfield is used to pass a single parameter from the exception to the exception handler. For exceptions with the same cause code, this parameter field is used to indicate the exact violation. Each bit in the parameter field is used to identify an exception. When two or more exceptions occur simultaneously, the corresponding bits are set in the parameter field.

For the TRAP exception, this field contains the zero-extended 6-bit immediate value specified by the operand of the TRAP instruction. For the [Privilege Violation, Disabled Extension](#) exception, this field contains the zero-extended 5-bit number of the disabled extension group that was accessed.

When an Actionpoint is hit, the parameter contains the number of the Actionpoint that triggered the exception.

The parameter field can also be used by extension instructions that raise exceptions.

6.4.2.6 Exception Cause Register

The Exception Cause register (see [Exception Cause Register, ECR](#)) provides an exception handler access to information about the source of the exception condition. The value in the Exception Cause register is made up from the Interrupt Code, Vector Number, Cause Code, and Parameter, as shown in [Figure 5-43](#) on page 186.

For example, the TRAP exception has the following values:

- Interrupt Code: 0x00
- Vector Name: EV_Trap
- Vector Number: 0x9

- Vector Offset: 0x24
- Cause Code: 0x00
- Parameter: *nn*

The exception cause register value for TRAP is 0x0900*nn*.

6.4.3 Exception Types and Priorities

Multiple exceptions can be associated with a single instruction, but only one exception can be handled at a time. Remaining exceptions are considered when the instruction is restarted after the first exception handler has completed. This process continues until no further exceptions remain, or a non-recoverable exception is encountered.

Interrupts and exceptions are evaluated with the following priority:

1. Reset
2. Machine Check, Non-maskable imprecise exception double Fault
3. Machine Check: Non-maskable imprecise exception
4. Machine Check, Double Fault
5. Machine Check, Fatal TLB errors
 - a. Invalid TLB command
 - b. Overlapping TLB entries
 - c. Illegal Overlapping MPU entries
6. Interrupt
7. TLB Fault on Instruction Fetch
 - a. Machine Check, Memory Error on ITLB Access
 - b. ITLB miss
8. Protection Violation on Instruction Fetch (MMU or MPU)
9. Memory Error on instruction fetch:
 - a. Bus error accessing external instruction memory
 - b. Instruction fetch from unpopulated ICCM region
 - c. Instruction fetch spanning multiple MPU regions
 - d. Instruction fetch spanning multiple instruction memory targets
10. Privilege Violation, Actionpoint hit on Instruction fetch
11. Machine Check, Internal Instruction Memory Error
12. Instruction Error
 - a. Illegal instruction exception
 - b. Illegal instruction sequence exception
13. Privilege Violation

- a. Access to kernel resource in user mode
 - i. Kernel -only instruction or register violation
 - ii. Access to a secured resource in normal mode
- b. Access to a secured resource in normal mode
 - i. Normal-mode-to secure-mode return with the wrong return type
 - ii. Normal-mode-to-secure-mode with unexpected user or kernel mode
 - iii. Interrupt or exception return fetch from the wrong mode
 - iv. Attempt to access a secure APEX feature from normal mode
 - v. SID violation on a secure resource
- c. APEX extension group
 - i. APEX extension group – User mode access disabled by XPU
 - ii. APEX extension group – kernel only extension violation
- d. Attempt to access secure APEX feature from normal mode
- e. SID violation on access to APEX feature
- f. Attempt to halt core from normal mode
- g. Disabled extension group
- 14. Memory error on XY Access
 - a. Illegal AGU memory target address
 - b. Unaligned AGU address spanning boundary
 - c. 64-bit AGU destination address is not 32 bit aligned
- 15. Extension Instruction Exception - requested by extension instruction
- 16. Misaligned data memory access
- 17. TLB fault on data access
 - a. Machine check, memory error on DTLB access
 - b. DTLB miss
- 18. Protection Violation
 - a. Protection Violation on data access (MMU, MPU, Code Protection, Stack Checking, or secure)
 - b. Non-sequential execute target address MPU violations (overlap region, illegal secure to normal transition)
- 19. Memory error on data access:
 - a. Data access to unpopulated DCCM region or absent memory
 - b. Data access spanning multiple data memory targets
- 20. Machine check, internal data memory error
- 21. Divide by zero exception when STATUS32[DZ] = 1
- 22. Trap or software interrupt (TRAP_S, SWI or SWI_S instructions)

23. Bus error on data access to external data memory (or ICCM)
24. Actionpoint hit on Auxiliary register or Memory-access — an imprecise exception

The raising of Actionpoint exceptions on memory accesses or auxiliary registers accesses may be delayed beyond the point at which the triggering instruction completes, possibly allowing several subsequent instructions to be completed before the Actionpoint exception is taken. For further details on implementation-specific Actionpoint delays, see the related *ARCV2-based processor Databook*.

Table 6-6 describes the exception vectors, exception cause codes, and exception parameters.

6.4.3.1 APEX Exception Priority Levels

The APEX exceptions are raised in the following categories:

1. APEX Extension Group – User mode access is disabled by XPU
2. APEX Extension Group – Access to privileged feature

Within these APEX exceptions, the priority list for APEX events is as follows:

1. Instruction extensions
2. Extension condition codes
3. An extension core register in a non-load destination context
4. An extension core register is used to return a value loaded from memory
5. An extension core register is used as the first source register (the B operand register)
6. An extension core register is used as the second source register (the C operand register)
7. Extension auxiliary register access

6.4.3.2 Exception Vectors and Cause Codes

Table 6-6 indicates the ECR values when an exception is raised. If simultaneous violations occur, the Parameter field in ECR has one bit set for each of those exceptions. For example, if there are simultaneous Code Protection, Stack Checking memory write violations, the Parameter field reads 3 (bit 0 is set for code protection and bit 1 is set for stack checking).

Table 6-7 on page 275 provides a summary of the values placed in the EFA and ERET registers for different exception types.

Table 6-6 Exception Vectors and Cause Codes

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Reset	Reset	0x00	0x00	0x00	0x00	0x000000	Always
Bus Error from Instruction Memory	Memory Error	0x04	0x01	0x00	0x0u	0x01000u	Always

Table 6-6 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Bus Error from Data Memory	Memory Error	0x04	0x01	0x10	0x0u	0x01100u'	
Instruction Fetch spanning multiple instruction memory targets	Memory Error	0x04	0x01	0x02	0x00	0x010200	
Instruction Fetch spanning multiple MPU regions	Memory Error	0x04	0x01	0x03	0x00	0x010300	Always
Data access spanning multiple data memory targets	Memory Error	0x04	0x01	0x12	0x00	0x011200	
NVM write or erase programming failure	Memory Error	0x04	0x01	0x13	0x00	0x011300	
NVM address out of range (instruction fetch or data access) only when NVM is configured	Memory Error	0x04	0x01	0x14	0x00	0x011400	
XY access to illegal target memory	Memory Error	0x04	0x01	0x21	0x00	0x012100	
XY access spanning multiple targets	Memory Error	0x04	0x01	0x22	0x00	0x012200	
XY 64-bit destination is not 32-bit aligned	Memory Error	0x04	0x01	0x23	0x00	0x012300	
Illegal Instruction	Instruction Error	0x08	0x02	0x00	0x00	0x020000	
Illegal Instruction Sequence	Instruction Error	0x08	0x02	0x01	0x00	0x020100	
Illegal instruction execution from S to NS (branch w/o link)	Instruction Error	0x08	0x02	0x10	0x00	0x021000	
Illegal instruction sequence, linear execution from S to NS	Instruction Error	0x08	0x02	0x11	0x00	0x021100	
Illegal instruction execution from NS to S (non-linear)	Instruction Error	0x08	0x02	0x12	0x00	0x021200	
Illegal instruction sequence, linear execution from NS to S	Instruction Error	0x08	0x02	0x13	0x00	0x021300	

Table 6-6 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Double Fault	EV_MachineCheck	0x0C	0x03	0x00	0x00	0x030000	
Overlapping TLB Entries	EV_MachineCheck	0x0C	0x03	0x01	0x00	0x030100	
Fatal TLB Error	EV_MachineCheck	0x0C	0x03	0x02	0x00	0x0302nn	
Internal Memory Error on Instruction Fetch	EV_MachineCheck	0x0C	0x03	0x04	0xrr	0x0304rr	
Internal Memory Error on Data Access	EV_MachineCheck	0x0C	0x03	0x05	0xrr	0x0305rr	
Illegal Overlapping MPU Entries	EV_MachineCheck	0x0C	0x03	0x06	0x00	0x030600	Always
Illegal Overlapping MPU entries (jump and branch target)	EV_MachineCheck	0x0C	0x03	0x06	0x01	0x030601	Always
Secure Vector Table not located in Secure memory	EV_MachineCheck	0x0C	0x03	0x10	0x00	0x031000	Always
NSC jump table not located in Secure memory	EV_MachineCheck	0x0C	0x03	0x11	0x00	0x031100	Always
Secure handler code not located in secure memory	EV_MachineCheck	0x0C	0x03	0x12	0x00	0x031200	Always
NSC target address not located in Secure memory	EV_MachineCheck	0x0C	0x03	0x13	0x00	0x031300	Always
Non-Maskable imprecise exception	EV_MachineCheck	0x0C	0x03	0x14	0x00	0x031400	
Non-maskable imprecise exception double fault	EV_MachineCheck	0x0C	0x03	0x14	0x01	0x031401	
Instruction Fetch TLB Miss	EV_ITLBMiss	0x10	0x04	0x00	0x00	0x040000	
Data TLB Miss on read	EV_DTLBMiss	0x14	0x05	0x01	0x00	0x050100	
Data TLB miss on write	EV_DTLBMiss	0x14	0x05	0x02	0x00	0x050200	
Data TLB miss on RMW	EV_DTLBMiss	0x14	0x05	0x03	0x00	0x050300	
Instruction Fetch Protection Violation in MPU	EV_ProtV	0x18	0x06	0x00	0x04	0x060004	

Table 6-6 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Instruction Fetch Protection Violation in OLM	EV_ProtV	0x18	0x06	0x00	0x08	0x060008	
Instruction Fetch Protection Violation on invalid Secure/Normal transition	EV_ProtV	0x18	0x06	0x00	0x24	0x060024	Always
Instruction-fetch invalid SID value (multiple bits set)	EV_ProtV	0x18	0x06	0x00	0x44	0x060044	Always
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in code protection scheme (parameter code 0x01)	EV_ProtV	0x18	0x06	0x01	0x01	0x060101	
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in stack checking scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x01	0x02	0x060102	
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in MPU (parameter code 0x04)	EV_ProtV	0x18	0x06	0x01	0x04	0x060104	
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in OLM (parameter code 0x08)	EV_ProtV	0x18	0x06	0x01	0x08	0x060108	
Memory Read Protection Violation from Secure MPU	EV_ProtV	0x18	0x06	0x01	0x24	0x060124	Always
Memory Read Protection Violation from Secure MPU with SID Mismatch	EV_ProtV	0x18	0x06	0x01	0x44	0x060144	Always
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in code protection scheme (parameter code 0x01)	EV_ProtV	0x18	0x06	0x02	0x01	0x060201	

Table 6-6 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in stack checking scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x02	0x02	0x060202	
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in MPU (parameter code 0x04)	EV_ProtV	0x18	0x06	0x02	0x04	0x060204	
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in OLM (parameter code 0x08)	EV_ProtV	0x18	0x06	0x02	0x08	0x060208	
Memory Write Protection Violation from NVM	EV_ProtV	0x18	0x06	0x02	0x10	0x060210	
Memory Write Protection Violation from Secure MPU	EV_ProtV	0x18	0x06	0x02	0x24	0x060224	Always
Memory Write Protection Violation from Secure MPU with SID Mismatch	EV_ProtV	0x18	0x06	0x02	0x44	0x060244	Always
Memory Read-Modify-Write (EX) protection violation in code protection (parameter code 0x01)	EV_ProtV	0x18	0x06	0x03	0x01	0x060301	
Memory Read-Modify-Write (EX) protection violation in stack checking protection scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x03	0x02	0x060302	
Memory Read-Modify-Write (EX) protection violation in MPU scheme (parameter code 0x04)	EV_ProtV	0x18	0x06	0x03	0x04	0x060304	
Memory Read-Modify-Write (EX) protection violation in OLM scheme (parameter code 0x08)	EV_ProtV	0x18	0x06	0x03	0x08	0x060308	

Table 6-6 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Memory Read-Modify-Write Protection Violation from NVM	EV_ProtV	0x18	0x06	0x03	0x10	0x060310	
Memory Read-Modify-Write Protection Violation from Secure MPU	EV_ProtV	0x18	0x06	0x03	0x24	0x060324	Always
Memory Read-Modify-Write Protection Violation from Secure MPU with SID Mismatch	EV_ProtV	0x18	0x06	0x03	0x44	0x060344	Always
Normal Vector Table in Secure Memory	EV_ProtV	0x18	0x06	0x10	0x00	0x061000	Always
NS handler code located in S memory	EV_ProtV	0x18	0x06	0x11	0x00	0x061100	Always
NSC Table Range Violation	EV_ProtV	0x18	0x06	0x12	0x00	0x061200	Always
Action Point Hit, Instruction Fetch	EV_PrivilegeV	0x1C	0x07	0x02	0xnn	0x0702nn	
Privilege Violation	EV_PrivilegeV	0x1C	0x07	0x00	0x00	0x070000	
Disabled Extension	EV_PrivilegeV	0x1C	0x07	0x01	0xnn	0x0701nn	
Action Point Hit, Memory or Register	EV_PrivilegeV	0x1C	0x07	0x02	0xnn	0x0702nn	
N to S Return Using Incorrect Return Mechanism	EV_PrivilegeV	0x1C	0x07	0x10	0x01	0x071001	Always
N to S Return with Incorrect Operating Mode	EV_PrivilegeV	0x1C	0x07	0x10	0x02	0x071002	Always
IRQ/Exception Return fetch from wrong mode	EV_PrivilegeV	0x1C	0x07	0x10	0x03	0x071003	Always
Attempt to halt secure processor in NS mode	EV_PrivilegeV	0x1C	0x07	0x10	0x04	0x071004	Always
Attempt to access secure resource from normal mode	EV_PrivilegeV	0x1C	0x07	0x10	0x20	0x071020	Always
SID Violation on Resource Access (APEX / UAUX / key NVM)	EV_PrivilegeV	0x1C	0x07	0x10	0x40	0x071040	Always

Table 6-6 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Attempt to access Secure APEX feature from NS mode	EV_PrivilegeV	0x1C	0x07	0x13	0x20	0x071320	Always
SID violation on access to APEX feature	EV_PrivilegeV	0x1C	0x07	0x13	0x40	0x071340	Always
Software Interrupt	EV_SWI	0x20	0x08	0x00	0xnn	0x0800nn	
Trap	EV_Trap	0x24	0x09	0x00	0xnn	0x0900nn	
Extension Instruction Exception	EV_Extension	0x28	0x0A	mm	0xnn	0x0Ammnn	
Invalid Operation, Floating-point extension exceptions	EV_Extension	0x28	0x0A	0x00	0x01	0x0A0001	
Divide by Zero, Floating-point extension exceptions	EV_Extension	0x28	0x0A	0x00	0x02	0x0A0002	
Divide by zero, Integer exception	EV_DivZero	0x2C	0x0B	0x00	0x00	0x0B0000	
Misaligned data access	EV_Misaligned	0x34	0x0D	0x00	0x00	0x0D0000	
Reserved	-	0x3C	0x0F	-		-	

The value of 'u' in the parameter field of Bus Error exceptions is determined by the value of STATUS32[U] bit at the time that the instruction triggering the exception was committed. Hence, a User-mode instruction fetch that results in a bus error from external memory, raises an exception with ECR 0x010001. The same exception in Kernel mode has an ECR value of 0x010000.

A load/store access targeting ICCM may be treated in some implementations as an external memory access (external to the Data Memory interface); in this case, if a memory error such as uncorrectable ECC error is encountered in either a load or partial store (RMW) to ICCM, a Bus Error from Data Memory exception may be raised. See [“Bus Error on Data Access to External Data Memory”](#) on page 286.

An example of an extension is floating-point extension. For this extension, the following are the values of sub cause code and parameter:

mm = 0 — indicates floating-point extension exceptions

nn—indicates specific exception(s) raised

nn	=	1	— Invalid Operation floating-point exception
	=	2	— Divide by Zero floating-point exception

The rules for prioritizing exceptions are applied normally, when multiple MMU exceptions are raised on two adjacent pages accessed by non-aligned memory operations. That means that the highest priority exceptions are always taken in preference

to lower priority exceptions. If two exceptions of the same type occur, for different pages, from the same memory reference, then the exception triggered by the effective byte address (EBA) of the memory operation should be taken first. If a reference spans two pages, then the second address will be EBA+4.

The parameter codes (rr) are implementation dependent. See the processor databook for more information about the error parameter codes for this exception.

For SJLI instruction, this exception occurs after the SJLI instruction commits.

BRK/BRK_S instructions can halt the processor in the normal kernel mode. In that case, exception is not raised.

For a fatal TLB error, the parameters are as follows: nn==04 when ECC or parity errors occur in instruction cache data memory during instruction fetch, nn==05 when ECC or parity errors occur in instruction cache tag memory during instruction fetch, nn==06 when ECC errors occur in address during instruction fetch

Table 6-7 EFA and ERET Entries for Exception Types

Exception	EFA Entry	ERET Entry
Memory access	Address of data access	PC address
Other non-SecureShield instruction exceptions	PC address	PC address
Watchpoints	Undefined	Imprecise exception, implementation-dependent
Actionpoint on instruction fetch	PC address	PC address
Bus error caused by uncached load	Data-access address (physical)	PC address
Bus error caused by uncached store	Data-access address (physical)	Imprecise exception, implementation-dependent
Instruction-fetch-related memory exception/ECC parity	PC address	PC address
Register file ECC exception	PC address	PC address
Program Counter exception	Faulty PC address	Faulty PC address
Bus error reported during data-cache copy-back write	Physical address of the cache line being copied back	Imprecise exception, implementation-dependent
Memory data access ECC or parity error on CCM/data-cache	Data-access address	PC or Next PC
ECC or parity error on data-cache data RAM during copy-back (triggered by LD/ST/EX cache miss or auxiliary operation)	Physical address of the cache line	Imprecise exception, implementation-dependent
Memory access ECC or parity error on data-cache tag RAM, DCCM	Data-access address	PC address
ECC or parity error on data-cache tag RAM during copy-back or aux operation	address from tag RAM (erroneous)	Imprecise exception, implementation-dependent
Exception during micro-op prologue (no secure shield 2+2 mode are configured)	Vector table entry address for vector fetch or DMP address for data access	Unchanged since the last updated value at prologue entry
Double-bit ECC error or parity error	PC address	PC address

SecureShield 2+2 Mode Exceptions

Table 6-7 EFA and ERET Entries for Exception Types (Continued)

Exception	EFA Entry	ERET Entry
Instruction fetch spanning multiple MPU regions	PC address	PC address
Illegal overlapping MPU entries	PC address	PC address
Secure vector table in non-secure memory space	PC address (vector-table address)	PC address (vector-table address)
NSC jump table not located in secure memory	PC address of SJLI instruction	PC address of SJLI instruction
Secure handler code not located in secure memory	PC address (vector-table address)	PC address (vector-table address)
NSC target address not located in secure memory	PC address (vector-table address)	PC address (NSC-table address)
Instruction fetch with invalid secure-normal transition violation on invalid secure-normal transition	PC address	PC address
Secure-mode violation on memory read (MPU) 0x060124	Data-access address	PC address
SID mismatch on memory read (MPU) 0x060144	Data-access address	PC address
Secure-mode violation on memory write (MPU) 0x060224	Data-access address	PC address
SID mismatch on memory write (MPU) 0x060244	Data-access address	PC address
Secure-mode violation on memory read-modify-write (MPU) 0x060324	Data-access address	PC address
SID mismatch on memory Read-modify-write (MPU) 0x060344	Data-access address	PC address
Normal vector table in secure memory	PC address (vector-table address)	PC address (vector-table address)
Normal handler code located in secure memory	PC address (vector-table address)	PC address (vector-table address)
NSC table range violation	PC address of SJLI instruction	PC address of SJLI instruction
Wrong return type (normal to secure with wrong return type) 0x071001	PC address	PC address
Function return fetch from wrong user or kernel mode 0x071002	PC address	PC address

Table 6-7 EFA and ERET Entries for Exception Types (Continued)

Exception	EFA Entry	ERET Entry
Function/, IRQ, or exception return fetch from wrong secure or normal mode 0x071003	PC address (undefined on normal-to-secure function return)	PC address (undefined on normal-to-secure function return)
Attempt to access secure resource from normal mode	PC address	PC address
SID violation on resource access (APEX, UAUX, key, NV_ICCM)	PC address	PC address
Exception during micro-op prologue (secure shield 2+2 mode are configured)	Vector table entry address	Vector table entry address

6.4.3.3 Reset

A reset is an external reset signal that causes the ARCV2-based processor to perform a *hard* reset. After reset, various internal states of the ARCV2-based processor are preset to their initial values, that is:

1. Interrupts are disabled.
2. The status register flags are cleared.
3. The loop count, loop start, and loop end registers are cleared.
4. The scoreboard unit is cleared.
5. The pending-load flag is cleared.

When `-sec_modes_option==true`, program execution begins at the address referenced by the four byte reset vector located at the interrupt vector base address, which is the first entry (offset 0x00) in the vector table. After reset, the vector table base address is set by configuration parameter `INT_VECTOR_BASE_S` (x1024) and reflected as a read-only value in build register [Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD](#). The core registers are not initialized except loop count (which is cleared). The Reset value of vector base register determines the Reset vector address. This reset value is configured at build time through the `-intvbase_preset_s` configuration option. When `-sec_modes_option==false`:

Program execution begins at the address referenced by the four byte reset vector located at the interrupt vector base address, which is the first entry (offset 0x00) in the vector table. After reset, the vector table base address is set by configuration parameter `INTVBASE_PRESET` (x1024) and reflected as a read-only value in build register [Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD](#). The core registers are not initialized except loop count (which is cleared). The Reset value of vector base register determines the Reset vector address. This reset value is configured at build time through the `INTVBASE_PRESET` parameter. The processor begins execution after reset by fetching the instruction at the address referenced by the reset vector.

A soft reset (an indirect jump to the Reset vector) does *not* preset any of the internal states of the ARCV2-based processor. For more information about how to do a soft reset, see [“Pass Control to an Handler or Perform a Soft Reset”](#) on page 265.

When an MMU is supported and configured in a processor, the `VECBASE_AC_BUILD.ADDR` must be set to an address in the non-translated memory space. This address is within the upper 2 GBytes of the memory

space. This design prevents double-fault exceptions from occurring when accessing the vector table to service a first exception. Fetching the vectors is then not subject to TLB misses or other TLB exceptions.

6.4.3.4 Machine Check, Non-Maskable Imprecise Exceptions

A non-maskable imprecise exception is triggered when the `nmi` external signal is asserted high.

6.4.3.4.1 Machine Check, Non-Maskable Imprecise Exception, Double fault

A non-maskable imprecise double-fault exception is taken in the following situations:

- The core is handling an exception and a non-maskable imprecise exception is triggered.
- The core is handling a non-maskable imprecise exception and another exception is raised.

6.4.3.5 Machine Check, Double Fault

Exception detected with exception handler outstanding, as indicated by STATUS32[AE] bit set (see [Status Register, STATUS32](#)).

The following situations also raise a double fault exception:

- Exception accesses normal vector table in the secure memory
- Normal exception handler is located in the secure memory.

If either the outstanding exception or the exception causing a double fault is a secure exception, the double fault exception must be handled in the secure mode. The double fault exception is handled in the normal kernel mode only if both the outstanding exception and the exception causing a double fault are marked as normal exceptions. For double exception, if the first exception is a secure exception, the second exception will also be a secure exception. If the first exception is a normal exception, and if the second exception is a secure exception, the exception status registers do not need protection because the saved information is normal.

6.4.3.6 Machine Check, Fatal TLB Error

- Any fatal error in the TLB or its memories (such as an uncorrectable parity or ECC error).

6.4.3.7 Machine Check, Overlapping TLB Entries:

- Multiple matches for an address lookup in the TLB. If this exception occurs in a user mode process, then the operating system can attempt to clean up and potentially restart the process. If it occurs in kernel mode, this is general fatal.
- **Illegal Overlapping MPU entries:** Illegal overlapping MPU entries, jump and branch instruction or micro-op sequence target is checked for exception only if the jump/branch is taken and the instruction is not a normal to secure function return. The current instruction MPU entries overlap exception has higher priority than the jump/branch target exception.

**Note**

For SJLI instruction, this exception occurs after the SJLI instruction commits.

6.4.3.8 Protection Violation, Instruction Fetch

An instruction fetch is fetched without the execute permission set (MMU, MPU, code protection, or stack checking).

The following cases raise secure instruction fetch exceptions:

- I-fetch invalid S/N transition (MPU): If a linear execution caused this exception, the exception is raised on the instruction that crosses into another MPU region. If a nonlinear branch or jump causes this exception, the exception is raised on the branch or jump instruction. If the branch or jump instruction has a delay slot instruction, the delay slot instruction is not executed due to this exception. Following instructions are checked for this exception:

When branch is taken	■ JL/JL_S<.d> ■ J/J_S<.d> ■ Jcc<.d>/JEQ_S ■ leave_s [..., pcl] ■ BL<.cc><.d> ■ BL_S ■ B<.d> ■ B_S ■ B<.cc><.d> ■ Bcc_S ■ BBIT0 ■ BBIT1 ■ DBNZ<.d>
When loop condition is false	LPcc

- Instruction fetch invalid on SID value.
- Secure mode violation on memory read.
- SID mismatch memory read.
- Secure mode violation on memory write.
- SID mismatch on memory write.
- Secure mode violation on memory-read-modify-write.
- SID mismatch on memory-read-modify-write.
- Secure Normal vector table is located in the secure memory;
- Normal interrupt or exception handler is located in the secure memory or the secure interrupt/exception handler code is in the normal memory: This exception is raised on last jump micro-op of the prologue sequence, and the exception PC (ERET) is the address of the entry in the vector table.

- Normal callable function table range violation: this exception is raised on the last jump micro-operation of the sequence, and the exception PC is the NSC table entry.

6.4.3.9 Memory Error on Instruction Fetch

6.4.3.9.1 Bus Error Accessing External Instruction Memory

A bus error exception is raised when an error occurs during a memory access that occurs externally to the unit from which it originated. For example, if an error occurs during an instruction fetch that does not access ICCM or Instruction cache, a bus error is reported. It is the responsibility of the target memory device to detect and report errors, via the appropriate bus interface. These bus errors may be triggered by parity or ECC errors within the external memory devices.

Bus error exceptions on external instruction memory accesses are always reported precisely.

6.4.3.9.2 Memory Error Exception, Access Spanning Multiple Memory Targets

A memory error exception is raised when an instruction fetch spans two regions that are designated as different memory targets. In this context, a memory target refers to the following types of memory region:

- ICCM
- External memory

**Note**

The DCCM and the peripheral Memory are not visible to the instruction fetch unit.

6.4.3.10 Memory Error Exception, Access Spanning Multiple MPU Regions

A memory error exception is raised when access span multiple MPU regions.

If an instruction fetch spans MPU regions and these multiple regions have different secure attributes, the region spanning cause code takes priority.

6.4.3.11 Privilege Violation, Action-Point Hit Instruction Fetch

Action point hit triggered by instruction fetch. The parameter field (nn) gives the number of the action point that triggered the exception.

- 0x00 = AP0
- 0x01 = AP1
- 0x02 = AP2
- 0x03 = AP3
- 0x04 = AP4
- 0x05 = AP5
- 0x06 = AP6
- 0x07 = AP7

6.4.3.12 Machine Check, Internal Instruction Memory Error

An uncorrectable memory error is detected in one of the internal memories accessed during an instruction fetch. Typically, these errors include ECC double errors, or parity errors, that are detected on instruction fetches from ICCM or Instruction Cache. Memory errors detected during incorrectly speculated accesses are ignored.

For an internal instruction memory error exception, the EFA contains the address of the instruction that contains the unrecoverable error. For an internal data memory error exception the EFA register contains the address of the data item address for which the unrecoverable error occurred.



Note

The parameter codes for these errors are implementation-dependent. See the processor databook for information on the ECR codes.

6.4.3.13 Instruction Error

6.4.3.13.1 Illegal Instruction

If an invalid instruction is fetched that the ARCV2-based processor cannot execute, an Illegal Instruction exception is caused. Any use of an unimplemented instruction, condition code, core register, or auxiliary register raises an Illegal Instruction exception. This exception is also raised if a jump or branch instruction with a delay slot specifies long-immediate data as the target address operand.

A predicated instruction always raises an Illegal Instruction exception if the instruction is encoded illegally, regardless of whether its predicate is true or false. In contrast, a correctly encoded instruction that accesses illegal operands, does not raise an Illegal Instruction exception, if it is predicated and its predicate is false. The false predicate means that the illegal operand is not actually accessed.

Note that a conditional branch instruction with false condition (a not-taken branch) also raises an Illegal Instruction exception if it is incorrectly encoded. Further, a BRcc or BBITn instruction that accesses illegal registers always raises an Illegal Instruction exception, even if its condition turns out to be false; its source registers need to be read to determine the condition, and therefore the operands of BRcc and BBITn are always accessed.

If an instruction specifies a L IMM source operand, and the instruction executes in a delay-slot context (that is, when STATUS32.DE is set to 1), the L IMM value is considered to be unpredictable. Such instructions do not raise an Illegal Instruction exception on the basis of the illegality of the L IMM value itself. However, they raise an Illegal Instruction Sequence exception due to the use of a L IMM source operand by an instruction that is in a delay slot.

For example, consider an AEX.PNZ instruction that accesses an undefined auxiliary register, specified using a core-register operand, when the Z flag is set. In this case the .PNZ predicate is false, and therefore the AEX.PNZ instruction does not raise an Illegal Instruction exception.

Consider also the example of an unconditional AEX instruction, which appears in a delay slot and accesses an undefined auxiliary-register address specified as a L IMM operand. Normally, an access to an unimplemented auxiliary register triggers an Illegal Instruction exception. However, a L IMM auxiliary register address in a delay-slot context is unpredictable and therefore does not trigger an Illegal Instruction exception. This AEX instruction instead raises an Illegal Instruction Sequence exception, due to the presence of its L IMM operand in a delay slot context, as described in [Illegal Instruction Sequence](#) section.

An illegal instruction error can also be caused by an invalid MMU command (SR with invalid command code to TLB Command register). An SR instruction that uses an unimplemented source register, or otherwise causes an Instruction Error exception, does not reach the stage of having its credentials checked and therefore does not raise a TLB error even though the command value is undefined.

6.4.3.13.2 Illegal Instruction Sequence

This exception is raised when an Illegal Instruction Sequence is attempted. This exception occurs in the following cases:

- When any jump or branch instruction straddles the loop end position such that:
 - The jump or branch instruction is in the last instruction position of the loop
 - The executed delay slot is outside the loop
- When any of the following instructions are attempted in the delay slot of a taken jump or branch:
 - Another jump or branch instruction (Bcc, BLcc, Jcc, JLcc, and so on)
 - A Loop instruction (LPcc)
 - A return from interrupt instruction (RTIE)
 - Any instruction with long-immediate data as a source operand
 - An ENTER_S or LEAVE_S instruction
 - An EI_S instruction
- When a LImm is present in the delay-slot instruction for any of the branch or jump instructions that set BLINK (BL.D, BLcc.D, JL.D, or JLcc.D)
- When any of the following instructions are attempted in the execution slot of an EI_S instruction:
 - Another EI_S instruction
 - Any jump or branch instruction (Bcc, BLcc, Jcc, JLcc)
 - A Loop instruction (LPcc)
 - A return from interrupt instruction (RTIE)
 - An ENTER_S or LEAVE_S instruction



Note

The ES bit of the STATUS32 auxiliary register indicates when the processor is in the execution slot of an EI_S instruction. The DE bit of the STATUS32 auxiliary register indicates when the processor is in the delay slot of a taken jump or branch.

- When the following instructions are prohibited in the last position (the instruction immediately before LOOP_END) of a zero-overhead loop body:
 - A branch or jump instruction (with or without a delay slot and either taken or not taken) (B, Bcc, Bcc_S, BR, BRcc, BRcc_S, BBIT0, BBIT1, BL, BLcc, BL_S, BI, BIH, J, Jcc, J_S, Jcc_s, J blink, JLcc, JL_S)

- The delay slot instruction of a taken branch or jump.

**Note**

If a branch is not taken it is not always possible to know that the delay slot instruction is actually a delay slot. That is the case if there is an intervening exception.

- ENTER_S, LEAVE_S, RTIE
- EI_S
- E-slot (the execute slot of an EI_S instruction).
- JLI_S
- LPcc

6.4.3.14 Privilege Violation

6.4.3.14.1 Secure Violations

When `-sec_modes_option==true`, the following cases raise the privilege violations:

- Secure Instruction or Register Access in the Normal Mode; an exception is raised when you attempt to read or write secure registers in the normal mode. This exception is also raised when you attempt to execute secure instructions in the normal mode.
- Attempts to halt the processor in the normal mode except when using the BRK/BRK_S instructions. You can halt a processor in the normal kernel mode using the BRK/BRK_S instructions.
- Attempts to access secure resources in the normal mode.
- SID violation on resource accesses for APEX, UAUX, and NV_ICCM.
- Mismatch in return type or mismatched return mode from a normal mode to the secure mode:
Mismatch in return type: The allowed return type from a normal mode to the secure more is defined by the SEC_STAT.NSRT field. This field specifies whether you can return from a function return or from an interrupt (RTIE). A privilege violation is raised if there is a mismatch of the return type (the actual return type) compared to the allowed return type specified in the SEC_STAT.NSRT field).
- Mismatch in return mode: The SEC_STAT.NSRU field specifies the return mode from a normal to a secure mode through a function call return or an RTIE instruction. If the return mode is a mismatch, a privilege violation exception is raised.
- IRQ/Exception return fetch mode: This exception is raised if there is a mismatch between the operating mode the IRQ/exception returns to and the return mode specified by the SEC_STAT.IRM (for interrupts) and the ERSEC_STAT.ERM (for exception) bits. This exception is raised on the last jump micro-op sequence of the epilog sequence, and the exception PC is the RTIE instruction. If this exception is taken on exception epilog, double fault will be raised.

6.4.3.14.2 Privilege Violation, Kernel/User Only Access

When the secure MPU is not configured, Kernel-only instruction, core register, or auxiliary register is accessed from the user mode.

When the secure MPU is configured, the kernel mode and user mode are completely independent. That is, user mode permissions do not imply kernel mode permissions when using the secure MPU. A privilege exception is raised when:

- In kernel mode you try to access resources with user-only permissions
- In user mode you try to access resources with kernel-only permissions

6.4.3.14.3 Privilege Violation, Disabled Extension

Disabled instruction or register is accessed. The parameter field (nn) gives the group number (0-31) of the disabled extension.

6.4.3.15 Extension Instruction Exception

This exception is triggered when an extension instruction requires that an exception be taken (for example, floating-point extensions need to generate many different types of exceptions).

The following are supplied by the extension instruction:

- mm = subcode
- nn = parameter

Floating-point Extension

mm = 0 — indicates floating-point extension exceptions

nn — indicates specific exceptions raised

nn	=	1 — Invalid Operation floating-point exception; the double-precision floating-point accelerator instructions (DADDx, DSUBx, DMULx, DEXCLx) do not raise this exception.
	=	2 — Divide by Zero floating-point exception; this exception is raised only by the hardware divider. A software divide implementation does not raise this exception.

6.4.3.16 Misaligned Data Access

A misaligned data access raises an EV_Misaligned exception. An access is considered misaligned if the address is not an integer multiple of the operand size and non-aligned references are not supported by the configured core or the STATUS32.AD bit is set to 0. For more information about the rules governing non-aligned memory references, see [Data Layout in Memory](#).

If a LD or ST using an unaligned AGU-based access straddles a modulo wrap boundary in which the buffer size is not a multiple of 16 bytes, a Misaligned memory access exception is raised: vector number 0x0D, vector offset 0x34, cause code 0x10, parameter 0x00. In this case, the exception fault address (EFA) register contains the address that triggered the exception.

6.4.3.17 TLB Fault on Data Access

6.4.3.17.1 Data TLB Miss

Data TLB miss caused by a LD, ST, PUSH_S, POP_S, or EX instruction. TLB lookup cannot locate an entry for the supplied virtual address

6.4.3.17.2 Data TLB Protection Violation

Data TLB protection violation caused by a LD, ST, PUSH_S, POP_S, EX, LLOCK, or SCOND instruction. This violation is caused when the attempted access does not match the permission bits for the page.

6.4.3.18 Instruction Fetch TLB Miss

Instruction fetch caused a miss in the μ ITLB and subsequently in the JTLB. Software must either copy an entry from the main page table to the JTLB or allocate the page in the main table then copy to JTLB.

6.4.3.19 Machine Check, Instruction Fetch Memory Error

A memory error is triggered by an instruction fetch. Memory errors triggered by incorrectly speculated accesses are ignored.

6.4.3.20 Protection Violation, Data Access

Memory Read, memory write, memory read-modify-write operations without the execute permission set (OLM, MPU, code protection, or stack checking).

Memory Read Protection Violation

Code attempts to load from a page that does not have the appropriate read permission bit set for the current core operating mode (kernel/user) or in the user mode code attempts to load to the untranslated region (when MPU is not configured in the processor).

Memory Write Protection Violation

Code attempts to store to a page that does not have the appropriate write permission bit set for the current core operating mode (kernel/user) or in the user mode code attempts to store to the untranslated region (when MPU is not configured in the processor).

Memory Read-Modify-Write Protection Violation

Code attempts an exchange operation to an address in a page that does not have the appropriate read and write permission bits set for the current CPU operating mode (kernel/user) or in the user mode code attempts to execute an exchange operation to the untranslated region (when MPU is not configured in the processor).

6.4.3.21 Memory Error on Data Access

6.4.3.21.1 Memory Error Exception, Access to Unpopulated Memory

A memory error exception is raised only when your processor configuration includes a non-volatile instruction closely coupled memory (NV_ICCM). This exception is raised when a data read or write is attempted within the NV_ICCM region, and any part of the memory location accessed is beyond the upper bound of the actual (NV_ICCM) memory. NV_ICCM is an asynchronous non-volatile memory that supports instruction fetch as well as load references but does not support store or execute access because of the long erase and program times on non-volatile memories.

6.4.3.21.2 Memory Error Exception, Access Spanning Multiple Memory Targets

A memory error exception is raised when a memory reference spans two regions that are designated as different memory targets. In this context, a memory target refers to the following types of memory region:

- ICCM
- DCCM
- Data Cache
- External memory (non-volatile memory)
- External memory (volatile memory)
- Peripheral memory
- Translated region
- Untranslated region

This exception cannot happen for aligned data accesses, but might happen for non-aligned data accesses: load and store instructions



Note

A reference spanning two regions that are both designated as external memory does not raise this exception. However, if one of the regions is assigned to DCCM and another is external memory, this exception is raised.

6.4.3.22 Machine Check, Internal Data Memory Error

An uncorrectable memory error is detected in one of the internal data memories accessed when performing a data read or write operation. Typically, these errors include ECC double errors, or parity errors, that are detected on load or store accesses to DCCM or Data Cache. Memory errors detected during incorrectly speculated accesses are ignored.

For an internal data memory error exception the EFA register contains the address of the data item address for which the unrecoverable error occurred.

6.4.3.23 Divide by Zero

This exception is triggered by a DIV, DIVU, REM, or REMU instruction that has a zero divisor operand when the STATUS32[DZ] bit is set to 1.

6.4.3.24 Trap or Software Interrupt

6.4.3.24.1 Software Interrupt

The SWI and SWI_S instructions use the EV_SWI vector. These instructions do not commit, but set the exception return address to be the address of the SWI or SWI_S instruction itself.

6.4.3.24.2 Trap

A TRAP_S instruction always commits, and the exception return address is the next instruction after the trap. This instruction is unlike all other exceptions, where the faulting instruction is aborted and the return address is that of the faulting instruction.

6.4.3.25 Bus Error on Data Access to External Data Memory

A bus error exception is raised when an error occurs during a memory access that occurs externally to the unit from which it originated. For example, if an error occurs during a data memory access to an uncached, non-DCCM location, a bus error is reported. Similarly, if an error occurs during an instruction fetch that does not access ICCM or Instruction cache, again a bus error is reported. It is the responsibility of the target

memory device to detect and report errors, via the appropriate bus interface. These bus errors may be triggered by accesses to unpopulated memory regions, or by parity or ECC errors within the external memory devices.

The raising of a bus error exception on an external data memory access may occur after the referencing instruction has committed. For example, if a cache block is written back to memory during cache replacement, and the write transaction experiences a bus error, the fault is caused by a memory address that is different from the one that is currently being accessed by the program.

Because precise exception handling is not possible for external data memory stores the exception return address (ERET) stored for an external data memory bus error is not guaranteed to be the address of the faulting instruction. In this case, the exception return address is the address of the next instruction to be executed in program sequence at the point in time when the exception is detected.

Successful recovery from an external data memory bus error is therefore not always possible.

Bus error exceptions on external instruction memory accesses are always reported precisely.

6.4.3.26 Data Cache Error

The EV_DCError exception is triggered when a non-cached memory operation, such as LD.DI or ST.DI, specifies a memory address that is valid in the data cache. The ability of an ARCV2-based processor to detect this condition is implementation specific. For more information, refer to the *ARCV2 ISA-based processor Databook*.

6.4.3.27 Privilege Violation, Action-Point Hit Memory or Register

This exception is triggered by Memory access, Core, or Auxiliary register access. The parameter field (nn) gives the number of the action point which triggered the exception.

6.4.4 Exception Detection

Exceptions are taken in strict program order. If more than one exception can be attributed to an instruction, the highest priority exception is taken and all others are ignored. Any remaining exception conditions may be handled when the faulting instruction is re-executed.

6.4.5 Effect of Exceptions and Interrupts on Operating Mode

Exceptions may be taken and handled from user mode or kernel mode and from interrupt service routines. An exception taken in an exception handler is a *double fault* condition and raises a fatal *machine check* exception.

All interrupts and exceptions cause an immediate switch to the kernel mode. The Memory Management Unit is not disabled on entry to an interrupt or exception handler, and the process-ID (ASID) register is not altered. All interrupts are disabled on entry to an exception handler.

6.4.6 Exception Entry

This section describes actions taken when an exception is detected. All addresses in this description are the logical addresses determined by the program.

1. The faulting instruction is dismissed.
 - No state changes caused by this instruction are committed

- All subsequent instructions that may be in a partial state of execution are also dismissed.
- All state changes associated with extension core registers or condition codes must also be prevented if an instruction is dismissed, so that on re-execution, the instruction functions correctly.

2. The exception-return register (see [Exception Return Address, ERET](#)) is loaded with the PC value used to fetch the faulting instruction.

If the exception is coerced using a [TRAP_S](#) instruction, the exception-return register (see [Exception Return Address, ERET](#)) is loaded with the address of the next instruction to be fetched after the TRAP instruction. This value is the architectural PC expected after the TRAP completes, so that pending branches and loops are taken into account.

3. The exception return status register (see [Exception Return Status, ERSTATUS](#)) is loaded with the contents of STATUS32 (see [Status Register, STATUS32](#)) used for execution of the faulting instruction. The [Exception Secure Status Register, ERSEC_STAT](#) is loaded with the contents of the [Secure Status Register, SEC_STAT](#).

- The value written to ERSTATUS (see [Exception Return Status, ERSTATUS](#)) is the last value committed to STATUS32 (see [Status Register, STATUS32](#)).
- If a delayed program-counter update is pending because the faulting instruction is in the delay slot of a taken branch or jump, the delay-slot bit is true. STATUS32[DE] = 1 (see [Status Register, STATUS32](#)).

4. The Exception Return Branch Target Address register ([Exception Return Branch Target Address, ERBTA](#)) is loaded with the current value of the Branch Target Address registers ([Branch Target Address, BTA](#)). If the STATUS32[DE] bit is set, or the STATUS32[ES] bit is set, BTA contains the target of a pending branch. On entry to an exception, the [Exception Return Status, ERSTATUS](#) register receives a copy of the pre-exception STATUS32 register and the [Exception Secure Status Register, ERSEC_STAT](#) is loaded with the contents of the [Secure Status Register, SEC_STAT](#). Together they hold sufficient contextual information to permit a pending branch to be taken on return from the exception. This mechanism is not affected by zero-overhead loops.

5. The exception cause register (see ECR) is loaded with a code to indicate the cause of the exception – see [Table 6-6](#).
6. When a memory access triggers an exception, including a double-fault exception, the exception fault address register (EFA) is loaded with the data item address that triggered the exception. For other faults, the EFA register is loaded with the PC value used to fetch the faulting instruction. For exceptions triggered by breakpoints, the EFA register is loaded with the PC value. Exceptions triggered by watchpoints are imprecise and the EFA register is thus undefined on their occurrence.
7. The CPU is switched into kernel mode STATUS32[U] = 0, and in the case of exceptions that are handled in secure mode, STATUS32[S] is set to 1.
8. Interrupts are disabled STATUS32[IE] = 0
9. The “active exception” flag is set. STATUS32[AE] = 1
10. ES (EI table instruction pending) bit is cleared, STATUS32[ES] = 0
11. SC (Stack Checking exception enabled) bit is cleared, STATUS32[SC] = 0
12. SSC (Secure stack checking exception enabled) bit is cleared, SEC_STAT[SSC]=0

13. DZ (Divide by Zero exception enabled) bit is cleared, STATUS32[DZ] = 0
14. The L bit in STATUS32 is set, disabling ZOL, STATUS32[L] = 1
15. The DE bit in the status register is cleared. STATUS32[DE] = 0
16. The Program Counter is loaded with the address of the appropriate exception vector. The appropriate exception vector is determined by the type of exception detected and the value in the interrupt and exception vector table base register.

The exception handlers must be able to save and restore all processor state that they alter during exception handling.

Saving the stack pointer provides a fixed location in the unmapped region of the address space for swapping the user-mode stack pointer with the exception stack pointer. The use of separate exception and interrupt stacks is a feature of many operating systems. Saving the stack pointer may also be necessary if the memory locations used for the user-mode stack for the faulting process do not have read and write privileges enabled for kernel mode.

6.4.6.1 Exceptions During Exception Entry

When an exception occurs during the fetching of an exception vector other than EV_MachineCheck (double fault), then the exception is converted into EV_MachineCheck (double fault) in the usual way.

When an exception occurs during the fetching of an exception vector for an EV_MachineCheck (double fault), a triple fault occurs. In such cases, the DEBUG.TF bit is set to 1, and the processor halts. The DEBUG.TF bit is available as an external output pin, if implemented, from each core, allowing external hardware to become aware of the fault and take any necessary action.

6.4.6.2 Regular Exception-Entry Code

```
Exception_enter() {
    U_saved = STATUS32.U
    /* update ERBTA, ECR, ERET, EFA; see "Exception Entry" on page 270. */
    if (STATUS32.U)
        AEX SP, AUX_USER_SP
    ERSTATUS = STATUS32
    STATUS32.U = 0
    STATUS32.{Z = U_saved, IE = 0, AE = 1, ES = 0, SC = 0, DZ = 0, L = 1, DE = 0}
    J INT_VECT_BASE[ECR.Vector_Num]
}
```

6.4.6.3 Exception-Entry Code When SecureShield 2+2 Mode is Configured

Exception-entry code when SecureShield 2+2 mode is configured is slightly more complex.

```
Exception_enter() {
    /* UOP pseudo code start: */
    Excpn_S = Excpn_S_mode(ECR.vector_num)
    U_saved = STATUS32.U
    Check_excpn_vt(ECR.Vector_Num, Excpn_S)
    /* update ERBTA, ECR, ERET, EFA; see "Exception Entry" on page 287. */

    ERSEC_STAT = SEC_STAT
    ERSEC_STAT.ERM = STATUS32.S
    ERSTATUS = STATUS32
}
```

```

SEC_STAT.SSC = 0
SEC_STAT.SUE = 0

Switch_sp(STATUS32.S, STATUS32.U, Excpn_S, 0)
Switch_duplicated_regs(Excpn_S)
/* Change mode to exception mode */
STATUS32.S = Excpn_S
STATUS32.U = 0

Check_excptn_handler_target(ECR.vector_num, Excpn_S)
STATUS32.{Z = U_saved, IE = 0, AE = 1, ES = 0, SC = 0, DZ = 0, L = 1, DE = 0}
if (Excpn_S) {
    J INT_VECT_BASE_S[ECR.Vector_Num]
}
else {
    J INT_VECT_BASE[ECR.Vector_Num]
}
/* UOP end, Processor start executing in mode of the fetching PC */

Set_mode_SID_by_MPU()
}

Check_excptn_handler_target(PI, s){
    /*s is the variable indicate current mode, 1: S, 0: N*/
    ENTRY_ADDR = (s ? INT_VECT_BASE_S : INT_VECT_BASE) + PI*4
    TARGET_ADDR = [ENTRY_ADDR]
    if (MPU_lookup(Target_ADDR).overlap){
        exception(EV_MachineCheck, ECR=0x030000)
    }
    if (TARGET_ADDR in N mode and s == 1){
        exception(EV_MachineCheck , ECR = 0x030000)
    }
    if (TARGET_ADDR in S mode and s == 0){
        exception(EV_MachineCheck, ECR = 0x030000)
    }
}

Check_excptn_vt(PI, s){
    /* s is the variable indicate current mode, 1: S, 0: N */
    PC = (s ? INT_VECT_BASE_S : INT_VECT_BASE) + PI*4
    if (MPU_lookup(PC).overlap) {
        exception(EV_MachineCheck, ECR=0x030000)
    }
    if (PC in N mode and s == 1){
        exception(EV_MachineCheck, ECR = 0x030000);
    }
    if (PC in S mode and s == 0){
        exception(EV_MachineCheck, ECR = 0x030000);
    }
    if (MPU_lkup(PC).sid has multi-bits set){
        exception(EV_MachineCheck, ECR = 0x030000);
    }
}

```

```

/* determine which S/N mode to switch to */
Excpn_mode(vector_num) {
    if (AUX_SEC_EXCEPT[vector_num])
        return 1
    else if (exception type must take in S mode)
        /* see Table 6-6 for exception vectors and cause codes */
        return 1
    else
        /* take exception in the original S/N mode */
        return STATUS32.S
}

```

Definitions of functions used in the example:

```

Switch_sp(ps, pu, s, u) {
    /* previous secure, previous user, current secure, current user */
    if (ps == s && pu == u) {
        return
    }
    Save_sp(ps, pu)
    Load_sp(s, u)
}

Switch_duplicated_regs(mode) {
    /* mode argument indicates the mode to switch to (mode == 0: N; mode == 1: S). Those
    operations are done automatically when current instruction/UOP change N/S mode, no UOP
    is issued as a result of duplicated register switch */
    LP_END(!mode) = LP_END          /* save ! mode version */
    LP_END = LP_END(mode)          /* load mode version */
    LP_START(!mode) = LP_START      /* save ! mode version */
    LP_START = LP_START(mode)      /* load mode version */
    LP_COUNT(!mode) = LP_COUNT      /* save ! mode version */
    LP_COUNT = LP_COUNT(mode)      /* load mode version */
    EI_BASE(!mode) = EI_BASE        /* save ! mode version */
    EI_BASE = EI_BASE(mode)        /* load mode version */
    LDI_BASE(!mode) = LDI_BASE      /* save ! mode version */
    LDI_BASE = LDI_BASE(mode)      /* load mode version */
    JLI_BASE(!mode) = JLI_BASE      /* save ! mode version */
    JLI_BASE = JLI_BASE(mode)      /* load mode version */
    BTA(!mode) = BTA               /* save ! mode version */
    BTA = BTA(mode)               /* load mode version */
}

Set_mode_SID_by_MPU() {
    /*set the mode and sid look up from fetch MPU */
    ifetch.SID = MPU_lookup(PC).SID
    STATUS32.S = MPU_lookup(PC).S
}

```

6.4.7 Exception Exit

After the exception handler completes its operations, the handler must restore the correct context for the task to continue execution. The [RTIE](#) instruction is used to return from exceptions.

When `-sec_modes_option==true`, the RTIE instruction returns to the operating mode specified by the `ERSEC_STAT[31]` field. If the instruction fetched from ERET has a different operating mode than current processor mode (specified by `ERSEC_STAT[31]`), a Privilege Violation exception is triggered.

The RTIE instruction uses the `AUX_IRQ_ACT` and AE bits of `STATUS32` to determine the set of link registers that are restored to PC, `STATUS32` and BTA, as shown in Table 6-8.

In Table 6-8, x signifies a Don't Care condition. A bit with a value of x means that whether the bit is 1 or 0, it does not affect the current operating interrupt priority of the processor.

Table 6-8 Exception and Interrupt Exit Modes

U	AE	AUX_IRQ_ACT	FIRQ_OPTION	Current Mode	RTIE Response	Link Registers Used
0	0	0000_0000_0000_0000	N/A	Kernel	Exception Exit	ERET, ERSTATUS, ERBTA
0	0	xxxx_xxxx_xxxx_xxx1	1	ISR P0	Interrupt Priority Level P0 Exit	ILINK, STATUS32_P0
0	0	xxxx_xxxx_xxxx_xxx1	0	ISR P0	Interrupt Priority Level P0 Exit	Restore from stack
0	0	xxxx_xxxx_xxxx_xx10	N/A	ISR P1	Interrupt Priority Level P1 Exit	Restore from stack
0	0	xxxx_xxxx_xxxx_x100	N/A	ISR P2	Interrupt Priority Level P2 Exit	Restore from stack
0	0	xxxx_xxxx_xxxx_1000	N/A	ISR P3	Interrupt Priority Level P3 Exit	Restore from stack
0	0	xxxx_xxxx_xxx1_0000	N/A	ISR P4	Interrupt Priority Level P4 Exit	Restore from stack
0	0	xxxx_xxxx_xx10_0000	N/A	ISR P5	Interrupt Priority Level P5 Exit	Restore from stack
0	0	xxxx_xxxx_x100_0000	N/A	ISR P6	Interrupt Priority Level P6 Exit	Restore from stack
0	0	xxxx_xxxx_1000_0000	N/A	ISR P7	Interrupt Priority Level P7 Exit	Restore from stack
0	0	xxxx_xxx1_0000_0000	N/A	ISR P8	Interrupt Priority Level P8 Exit	Restore from stack
0	0	xxxx_xx10_0000_0000	N/A	ISR P9	Interrupt Priority Level P9 Exit	Restore from stack
0	0	xxxx_x100_0000_0000	N/A	ISR P10	Interrupt Priority Level P10 Exit	Restore from stack
0	0	xxxx_1000_0000_0000	N/A	ISR P11	Interrupt Priority Level P11 Exit	Restore from stack

Table 6-8 Exception and Interrupt Exit Modes (Continued)

U	AE	AUX_IRQ_ACT	FIRQ_OPTION	Current Mode	RTIE Response	Link Registers Used
0	0	xxx1_0000_0000_0000	N/A	ISR P12	Interrupt Priority Level P12 Exit	Restore from stack
0	0	xx10_0000_0000_0000	N/A	ISR P13	Interrupt Priority Level P13 Exit	Restore from stack
0	0	x100_0000_0000_0000	N/A	ISR P14	Interrupt Priority Level P14 Exit	Restore from stack
0	0	1000_0000_0000_0000	N/A	ISR P15	Interrupt Priority Level P15 Exit	Restore from stack
0	1	xxxx_xxxx_xxxx_xxx	N/A	Exception	Exception Exit	ERET, ERSTATUS, ERBTA
1	-	xxxx_xxxx_xxxx_xxx	N/A	User	Privilege Violation	N/A

- AE and AUX_IRQ_ACT are all set to 0 for state changes from kernel mode (U bit is 0 in kernel mode), for example when scheduling a user mode task.
- If the AE bit is set, or AE and AUX_IRQ_ACT are all zero, the exception-exit sequence is followed. If AE is zero, and AUX_IRQ_ACT is set to true, the interrupt-exit sequence is followed. See description of the [RTIE](#) instruction for further details.
- The program counter is loaded with the exception return address from the ERET register, the contents of ERSTATUS are copied into STATUS32, the ERSEC_STAT[5:0] bits are copied into the SEC_STAT[5:0] register, and the contents of ERBTA (see [Exception Return Branch Target Address, ERBTA](#)) are copied into BTA (see [Branch Target Address, BTA](#)).
- If the delay-slot bit STATUS32[DE] (see [Status Register, STATUS32](#)) is set as a result, an unconditional delayed branch is triggered to the address contained in the branch target address register (BTA).
- The AE bits can be set to any desired values in kernel mode using the KFLAG instruction.
- If the STATUS32[DE] bit is set as a result of the RTIE instruction, the processor goes into a state where a branch with a delay slot is pending. The target of the branch is contained in the BTA register which is restored from the appropriate Exception Return BTA register (ERBTA).

Figure 6-4 Exception Exit in Normal Mode

```

Exception_exit() {
/* UOP pseudo code start: */
/* Assert (STATUS32.AE == 1) */
if (ERSTATUS.U)
    AEX SP, AUX_USER_SP
    STATUS32.U = ERSTATUS.U
    STATUS32 = ERSTATUS
    BTA = ERBTA
    J ERET

```

```

    }

```

Figure 6-5 Exception-Exit in SecureShield 2x2 Mode

```

Exception_exit() {
    /* UOP pseudo code start: */
    /* Assert (STATUS32.AE == 1) */
    ERM = ERSEC_STAT.ERM
    /* Switch SP and duplicated registers*/
    Switch_sp(STATUS32.S, 0, ERM, ERSTATUS.U)
    Switch_duplicated_regs(ERM)
    /* Change Mode to ERM and ERSTATUS.U*/
    STATUS32.U = ERSTATUS.U
    STATUS32.S = ERM
    /* Check return address is in correct mode, otherwise raise exception */
    /* If there's UOP pending and there's no dslo pending, mode change UOP */
    /* will be triggered immediately after exception return and handle the */
    /* switch to correct mode. There's no need to check mode match here */
    If (!(ERSEC_STAT.SUE & !ERSTATUS.DE))
        Check_jump_target(ERM, ERET)
    /* Restore SEC_STAT */
    SEC_STAT = ERSEC_STAT
    /* Clear ERM to prevent illegal return to S using rtie */
    ERSEC_STAT.ERM = 0
    /* Restore STATUS32 and other status */
    STATUS32_RESTORE = ERSTATUS
    STATUS32_RESTORE.S = ERM
    STATUS32 = STATUS32_RESTORE
    BTA = ERBTA
    J ERET
    /* UOP end, Processor start executing in mode of the fetching PC */
    Set_mode_SID_by_MPU()
}

```

Definitions of functions used in the example:

```

Switch_sp(ps, pu, s, u) {
    /* previous secure, previous user, current secure, current user */
    if (ps == s && pu == u) {
        return
    }
    Save_sp(ps, pu)
    Load_sp(s, u)
}

Check_jump_target(s, addr){
    if (MPU_lookup(addr).overlap){
        exception(EV_MachineCheck, ECR=0x030601)
    }
    /* is the variable indicate current mode, 1: S, 0: N */
    if (addr in N mode and s == 1){
        exception(EV_PrivilegeV, ECR = 0x071003)
    }
    if (addr in S mode and s == 0){

```

```

        exception(EV_PrivilegeV, ECR = 0x071003)
    }
}

Switch_duplicated_regs(mode) {
/* mode argument indicates the mode to switch to (mode == 0: N; mode == 1: S). Those
operations are done automatically when current instruction/UOP change N/S mode, no UOP
is issued as a result of duplicated register switch */
    LP_END(!mode) = LP_END          /* save ! mode version */
    LP_END = LP_END(mode)          /* load mode version */
    LP_START(!mode) = LP_START      /* save ! mode version */
    LP_START = LP_START(mode)      /* load mode version */
    LP_COUNT(!mode) = LP_COUNT      /* save ! mode version */
    LP_COUNT = LP_COUNT(mode)      /* load mode version */
    EI_BASE(!mode) = EI_BASE        /* save ! mode version */
    EI_BASE = EI_BASE(mode)        /* load mode version */
    LDI_BASE(!mode) = LDI_BASE      /* save ! mode version */
    LDI_BASE = LDI_BASE(mode)      /* load mode version */
    JLI_BASE(!mode) = JLI_BASE      /* save ! mode version */
    JLI_BASE = JLI_BASE(mode)      /* load mode version */
    BTA(!mode) = BTA               /* save ! mode version */
    BTA = BTA(mode)               /* load mode version */
}

Set_mode_SID_by_MPU() {
/*set the mode and sid look up from fetch MPU */
    ifetch.SID = MPU_lookup(PC).SID
    STATUS32.S = MPU_lookup(PC).S
}

```

6.4.8 Exceptions and Delay Slots

For the ARCV2-based processor, exceptions are supported for instructions in the delay slots of branches.

Example 6-9 Exception in a Delay Slot

```

J.D [blink]      ; Branch/Jump Instruction
LD fp, [sp, 24] ;
:                ;
MOV r0, 0        ; Target of the branch/jump

```

The ARCV2 architecture provides features for recovery from exceptions caused by instructions found in branch or jump delay slots.

The SEC_STAT[4] bit indicates whether a mode changing micro-op sequence is pending while a delay slot instruction is executing. It is used to determine that a micro-op sequence should execute after a breakpoint, sleep, or exception is handled after a branch has committed but before the branch-related delay slot instruction has committed. This bit is visible for processor read in the secure kernel mode and is read as zero for processor reads in other modes. Writes of 1 are ignored.

When an exception is detected on a delay-slot instruction, the return address stored on exception entry is normally the address of the instruction in the delay slot. This mechanism allows an exception handler to return to the delay-slot instruction of a taken branch, and for subsequent instructions to be executed starting at the branch target address.

In addition, this mechanism allows branch instructions that can change processor state to have delay slots, for example BRcc/BBITn/Jcc using auto-update extension core registers, or simply the BLcc instruction.

This mechanism removes many possible hazards that might result from not returning to a faulting instruction that was previously canceled, such as the possibility of TLB thrash or deadlock.

You must not specify a long-immediate source operand in the delay-slot instruction or in the instruction present in the delay slot. When such instructions are encountered, the processor may not fetch the long-immediate value from memory. Therefore, any memory protection mechanism applied to instruction fetch, for example MPU or MMU, is not guaranteed to detect a fetch fault due to the long-immediate operand access. For example, if a delay-slot and its illegal long-immediate data straddles an MPU region or an MMU page boundary, it is possible that only the first MPU region or MMU page is accessed. In such cases an Illegal Instruction Sequence exception may be raised rather than a privilege violation or TLB miss.

When a JL.D, JLcc.D, BL.D or BLcc.D instruction is executed, the size of the following delay-slot instruction is used in the calculation of the BLINK value. Normally, the size of each instruction is determined by its format (2 or 4 bytes) and the presence or absence of a long-immediate data operand.

However, a delay-slot instruction is not permitted to have a long-immediate operand. It is illegal for an instruction appearing in the delay-slot of a branch or jump instruction to have a long-immediate data operand (LIMM). The processor ignores any such LIMM operand specified by a delay-slot instruction, and consequently determines the size of the delay-slot instruction based solely on its major opcode. If the preceding branch or jump instruction sets the BLINK register (for example, JL.D or BL.D) then the BLINK value defined by that instruction is the address of its delay-slot instruction plus the size of its delay-slot instruction, but always excluding any illegal LIMM operand.

However, if the processor does not include any instructions encoded with the same major op-code as the unrecognized delay-slot instruction, the operands of that instruction are not recognized by the processor and it is unaware of the presence of a long-immediate source operand.

In such cases the size of the delay-slot instruction is determined solely by the size of the instruction format. This case could occur, for example, if an APEX instruction is issued when there is no APEX present.

A branch or jump with linkage and a delay slot always observes its delay-slot instruction, to allow it to compute the required BLINK value. However, if an exception is raised when fetching such a delay-slot instruction, then it is not possible for the branch or jump to compute the BLINK value. This is because the delay-slot instruction cannot be trusted and therefore its size cannot be determined.

When such a branch or jump observes its delay-slot instruction, and there is no fetch-related exception for that instruction, the branch instruction completes normally. Subsequently, a non-fetch exception may still be raised on the delay-slot instruction preventing it from completing. If the delay-slot instruction is later restarted, after resolving the condition that raised the exception, it is said to be an orphaned delay slot instruction. If the faulting delay-slot instruction is later resumed, it must be the same as the one that was observed by the parent branch when it executed.

Whenever a fetch-related exception is raised on a delay-slot instruction, whose parent branch or jump writes to BLINK, the exception is transferred to the parent branch. However, this does not happen to orphaned delay-slot instructions, as their parent branches will have already completed.

When a fetch-related exception is transferred to the parent branch, the ERET register is set to the address of the parent branch, whereas the EFA register is set to the address of the delay-slot instruction. The EFA register points the exception handler to the faulting instruction, and the ERET register defines the address to return to to resume execution from the parent branch."

6.4.9 Emulation of Extension Instructions

Illegal-instruction exceptions are triggered if an instruction references an unmapped extension operand. A handler for illegal-instruction exceptions must be able to do the following to emulate the function of an extension instruction.



Note

When an extension is present but disabled using the XPU register, the exception vector used is [Privilege Violation, Kernel/User Only Access](#).

- Get the address of the faulting instruction from the ERET register (see [Exception Return Address, ERET](#)).
- Disassemble the instruction sufficiently to determine whether the instruction must be emulated
- Perform the emulation function, and make any changes to processor state (real or emulated) that are required



Note

Any required changes to ZNCV flags must be made in the ERSTATUS register to be restored on exception return (see [Exception Return Status, ERSTATUS](#))

- Return to the next instruction *after* the emulated instruction. The return address might be one of the following (in order of priority):
 - ERBTA – exception branch target address if the faulting instruction was in the delay slot of a taken branch (see [Exception Return Branch Target Address, ERBTA](#))
 - [Loop Start Register, LP_START](#) if the faulting instruction was the last instruction in a zero-overhead loop, but not the last loop iteration ($ERET + \text{emulated\_instruction\_size} = LP\_END$, and $LP\_COUNT > 1$) (see [Exception Return Address, ERET](#)).
 - $ERET + \text{emulated\_instruction\_size}$ for normal, linear execution (see [Exception Return Address, ERET](#))

6.4.10 Emulation of Extension Registers and Condition Codes

A scheme similar to the one described in [Emulation of Extension Instructions](#) can be used to emulate extension registers and condition codes, also using the illegal-instruction exception.

7

Instruction Set Summary

This chapter provides an overview of the ARCV2 ISA. It presents an overview of how the ARCV2 ISA is encoded, explains the assembly syntax conventions used by the remainder of this document, and presents a summary of the instructions found in each functional instruction grouping. Information the encoding of each instruction can be found in [Chapter 8, “32-bit Instruction Formats Reference”](#) and [Chapter 9, “16-bit Instruction Formats Reference”](#), and full details of all aspects of each individual instruction can be found in [Chapter 10, “Instruction Set Details”](#).

Among the defining characteristics of ARC processors is their configurability and extensibility. The ARCV2 instruction set is therefore highly configurable. All ARCV2 processors support a minimum set of instructions and capabilities referred to as the basecase instruction set, plus an optional set of configured extensions. The available configuration options, and the additional instructions included with each option, are explained in [“ISA Options”](#) on page [108](#).

7.1 Instruction Set Encoding

The ARCV2 ISA provides a broad range of instructions, encoded in a mixture of 32-bit formats and 16-bit formats. The ARC instruction encoding defines a number of **top-level instruction formats**, each of which is identified by a 5-bit **major opcode** that is always positioned in the most-significant 5 bits of each instruction. Therefore, a 32-bit instruction locates the major opcode in bits [31:27], and 16-bit instructions locate the major opcode in bits [15:11]. The format of the remaining bits in each major opcode is specific to each top-level format.

Some of the most common instructions are encoded in both 32-bit and 16-bit formats, allowing compilers to encode instructions in a compact 16-bit format in many cases. The 16-bit formats have restricted numbers of operands, and each operand is able to refer to a sub-set of the available general-purpose registers. Through careful register allocation, compilers are able to encode instructions using 16-bit formats in many cases.

7.1.1 Top-level Instruction Formats

[Table 7-1](#) lists the 32 top-level instruction formats of the ARCV2 instruction set. For each format the table illustrates the type of instruction that is encoded in that format, indicates whether it is a 32-bit or a 16-bit format, and gives brief notes on those instructions.

All formats are named using a convention that 32-bit formats have an F32_ prefix and 16-bit formats have an F16_ prefix.

Table 7-1 Top-level Formats of the ARCV2 Instruction Set

Format	Description	Examples
F32_BR0	Branch conditionally Branch unconditionally far	Bcc s21 B s25
F32_BR1	Branch-and-link conditionally Branch-and-link unconditionally far Compare-and-branch	BLcc s21 BL s25 BRcc b,c,s9
F32_LD_OFFSET	Load using offset addresses	LD with register + offset addressing
F32_ST_OFFSET	Store using offset addresses	ST with register + offset addressing
F32_GEN4	ARC 32-bit basecase instructions	op a,b,c
F32_EXT5	ARC 32-bit extension instructions	op a,b,c
F32_EXT6	ARC 32-bit extension instructions	op a,b,c
F32_APEX	User 32-bit extension instructions	op a,b,c
F16_COMPACT	Move/Load with h-register	MOV_S g,h / LD_S R0-3,[h,u5]
F16_LD_ADD_SUB	Load/Add/Sub compact	LD_S.AS a,[b,c] / SUB_S a,b,c / ADD_S R0-1,b,u6
F16_LD_ST_1	Load/Store gp-relative Load-indexed	LD R1,[GP,s11] / ST R0,[GP,s11] LDI_S b,[u7]
F16_JLI_EI	Jump-and-link indexed Execute indexed	JLI_S EI_S
F16_LD_ADD_RR	Load/Add register-register	LD_S / LDB_S / LDH_S / ADD_S a,b,c
F16_ADD_IMM	Add/Subtract/Shift immediate	ADD_S / SUB_S / ASL_S / LSR_S c,b,u3
F16_OP_HREG	Move, Compare, Add with one h-register	MOV_S / CMP_S / ADD_S b,h / b,b,h
F16_GEN_OP	General ops and single ops	op_S b,b,c
F16_LD_WORD	Load word with short offset	LD_S c,[b,u7]
F16_LD_BYTE	Load byte with short offset	LDB_S c,[b,u5]
F16_LD_HALF	Load half-word with short offset	LDH_S c,[b,u6]
F16_LDX_HALF	Load signed half-word with short offset	LDH_S.X c,[b,u6]

Table 7-1 Top-level Formats of the ARCV2 Instruction Set

Format	Description	Examples
F16_ST_WORD	Store word with short offset	ST_S c,[b,u7]
F16_ST_BYTE	Store byte with short offset	STB_S c,[b,u5]
F16_ST_HALF	Store half-word with short offset	STH_S c,[b,u6]
F16_SH_SUB_BIT	Shift/Subtract/bit ops	op_S b,b,u5
F16_SP_OPS	SP-based Load/Store/Add/Subtract Function prolog, epilog	LD_S / LDB_S / ST_S / STB_S / ADD_S / SUB_S / PUSH_S / POP_S ENTER_S / LEAVE_S
F16_GP_LD_ADD	GP-based Load/Add	LD_S / LDH_S / LDB_S / ADD_S
F16_PCL_LD	PCL-based Load	LD_S b,[PCL,u10]
F16_MV_IMM	Move immediate	MOV_S b,u8
F16_OP_IMM	Add/Compare immediate	ADD_S / CMP_S b,u7
F16_BCC_REG	Branch conditionally on register Z/NZ	BRcc_S b,0,s8
F16_BCC	Branch conditionally	Bcc_S s10/s7
F16_BL	Branch and link unconditionally	BL_S s13
F32_GEN_OP64	ARC 64-bit instructions	op64 a,b,c
F32_VEC1	Vector DSP group 1	(defined in Vector DSP section)
F32_VEC2	Vector DSP group 2	
F32_VEC3	Vector DSP group 3	
F32_VEC4	Vector DSP group 4	
F32_VEC5	Vector DSP group 5	
F32_VEC6	Vector DSP group 6	

7.1.2 Instruction Set Profiles

The ARCV2 instruction set includes the concept of an **instruction set profile**. Each profile allows a small number of top-level formats to encode profile-specific sets of extension instructions that are designed for the application domains of that profile. This acknowledges the fact that a common set of instructions are applicable across all domain, although a unique use of every encoding is too restrictive for all potential use-cases.

The **Baseline Profile** defines a set of common top-level formats that are always included by default in all other profiles.

The **Compact Profile** is designed for general-purpose embedded controller and real-time systems, where both high performance and high code density are required. The Compact Profile extends the set of common formats with several 16-bit formats. This provides a broader set of 16-bit instruction encodings to maximize code density. Other profiles may use those major opcodes in different ways, to encode domain-specific instructions.

7.1.3 Assignment of Instruction Formats to Major Opcodes

Table 7-2 shows the assignment of the top-level instruction formats to the major opcodes of the ARCV2 instruction set architecture. Each major opcode is listed in the first column, and second columns lists the common encodings available to all profiles. The remaining columns show the assignment of major opcodes to encodings that are restricted to a particular profile.

Table 7-2 Assignment of top-level instruction formats to major opcodes in each instruction set profile

Major opcode	Common formats	Profile-specific formats	
	Baseline Profile	Compact Profile	Vector DSP Profile
0x00	F32_BR0		
0x01	F32_BR1		
0x02	F32_LD_OFFSET		
0x03	F32_ST_OFFSET		
0x04	F32_GEN4		
0x05	F32_EXT5		
0x06	F32_EXT6		
0x07	F32_APEX		
0x08	F16_COMPACT		
0x09	F16_LD_ADD_SUB		
0x0A	F16_LD_ST_1		
0x0B		F16_JLI_EI	
0x0C	F16_LD_ADD_RR		
0x0D	F16_ADD_IMM		
0x0E	F16_OP_HREG		
0x0F	F16_GEN_OP		
0x10	F16_LD_WORD		

Table 7-2 Assignment of top-level instruction formats to major opcodes in each instruction set profile

Major opcode	Common formats	Profile-specific formats	
	Baseline Profile	Compact Profile	Vector DSP Profile
0x11		F16_LD_BYTE	F32_VEC1
0x12		F16_LD_HALF	F32_VEC2
0x13		F16_LDX_HALF	F32_VEC3
0x14	F16_ST_WORD		
0x15		F16_ST_BYTE	F32_VEC4
0x16		F16_ST_HALF	F32_VEC5
0x17	F16_SH_SUB_BIT		
0x18	F16_SP_OPS		
0x19	F16_GP_LD_ADD		
0x1A	F16_PCL_LD		
0x1B	F16_MV_IMM		
0x1C	F16_OP_IMM		
0x1D	F16_BCC_REG		
0x1E	F16_BCC		
0x1F		F16_BL	F32_VEC6

7.1.4 32-bit Instruction Formats

Table 7-3 summarizes the field layout of the 32-bit formats in the ARCV2 ISA, arranged hierarchically by (a) the top-level format, (b) the number of operands in the format, and (c) the operand sub-formats.

Table 7-3 Summary of 32-bit instruction formats

32-bit Format Hierarchy			Major Opcode	Layout of Instruction Formats																																		
(a)	(b)	(c)	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				
F32_BR0		COND	>		0x0				S[10:1]								0 1		S[20:11]								N		Q[4:0]									
		UCOND_FAR																											R		S[24:21]							
F32_BR1	BL	COND	>		0x1				S[10:2]								0 1		S[20:11]								N				Q[4:0]							
																													UCOND_FAR	R		S[24:21]						
	BCC	REG_REG	>		B[2:0]				S[7:1]								1		S8		B[5:3]		C[5:0]															
																							REG_U6	U[5:0]				1		i[3:0]								
F32_LD_OFFSET			>		0x2				B[2:0]				S[7:0]								S8		B[5:3]				D			aa		ZZ		X		A[5:0]		
F32_ST_OFFSET		ST_REG	>		0x3				B[2:0]				S[7:0]								S8		B[5:3]				C[5:0]				D		aa		ZZ		0 1	
		ST_W6																									W[5:0]											
F32_GEN4	DOP	REG_REG	>		0x4				B[2:0]				00 01 10 11 aa		i[5:0] except when (i[5:3] == 110) or (i[5:0] == 101111)				F		B[5:3]				C[5:0]				A[5:0]									
																									REG_U6	U[5:0]												
																									REG_S12	S[5:0]				S[11:6]								
																									COND_REG	C[5:0]				0		Q[4:0]						
																									COND_U6	U[5:0]				1								
																									LD_REG	aa				1 1 0				ZZ		X		D
	SOP	REG_REG	>		0x4				B[2:0]				00 01		1 0 1 1 1 1				F		B[5:3]				C[5:0]				i[5:0]									
																									REG_U6	U[5:0]												
	ZOP	REG_REG	>		0x4				i[2:0]				00 01		1 0 1 1 1 1				F		i[5:3]				C[5:0]				1 1 1 1 1 1									
																									REG_U6	U[5:0]												
F32_EXT5, F32_EXT6, F32_APEX	DOP	REG_REG	>		0x5, 0x6, 0x7				B[2:0]				00 01 10 11		i[5:0] except when (i[5:0] == 101111)				F		B[5:3]				C[5:0]				A[5:0]									
																									REG_U6	U[5:0]												
																									REG_S12	S[5:0]				S[11:6]								
																									COND_REG	C[5:0]				0		Q[4:0]						
																									COND_U6	U[5:0]				1								
	SOP	REG_REG	>		0x5, 0x6, 0x7				B[2:0]				00 01		1 0 1 1 1 1				F		B[5:3]				C[5:0]				i[5:0] (except 111111)									
																									REG_U6	U[5:0]												
	ZOP	REG_REG	>		0x5, 0x6, 0x7				i[2:0]				00 01		1 0 1 1 1 1				F		i[5:3]				C[5:0]				1 1 1 1 1 1									
																									REG_U6	U[5:0]												

7.1.5 16-bit Instruction Formats

Table 7-4 summarizes the 16-bit compact instruction formats defined by the ARCV2 instruction set.

Table 7-4 Summary of 16-bit instruction formats

16-bit Formats		Major Opcode					Layout of Instruction Formats										
(a)	(b)	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F16_COMPACT	MOV	0x8					g[2:0]		h[2:0]			g[4:3]		0		h[4:3]	
	LD						U[5]					r[1:0]		U[3:2]			
F16_LD_ADD_SUB	REG_REG	0x9					b[2:0]		c[2:0]		i	0		a[2:0]			
	REG_U6								a[0]	U[5:3]			1				U[2:0]
F16_LD_ST_1	LD_ST_R01	0xA					S[10:5]					i	0		S[4:2]		
	LDI_S						b[2:0]		U[6:3]			1		U[2:0]			
F16_JLI_EI		0xB					i	U[9:0]									
F16_LD_ADD_RR		0xC					b[2:0]		c[2:0]		i[1:0]		a[2:0]				
F16_ADD_IMM		0xD					b[2:0]		c[2:0]		i[1:0]		U[2:0]				
F16_OP_HREG	HREG_REG	0xE					b[2:0]		h[2:0]			i[1:0]		0		h[4:3]	
	HREG_S3						S[2:0]							1			
F16_GEN_OP	DOP	0xF					b[2:0]		c[2:0]		i[4:0] != 00000						
	i[7:5] != 111								0 0 0 0 0								
	ZOP						i[8:6]		1 1 1								
F16_LD_WORD		0x10					b[2:0]		c[2:0]		U[2:0]						
F16_LD_BYTE		0x11															
F16_LD_HALF		0x12															
F16_LDX_HALF		0x13															
F16_ST_WORD		0x14															
F16_ST_BYTE		0x15															
F16_ST_HALF		0x16															
F16_SH_SUB_BIT		0x17					b[2:0]		i[2:0]		U[4:0]						
F16_SP_OPS		0x18					b[2:0]		i[2:0]		U[4:0]						
F16_GP_LD_ADD		0x19					i[1:0]		S[8:0]								
F16_PCL_LD		0x1A					b[2:0]		U[7:0]								
F16_MV_IMM		0x1B															
F16_OP_IMM		0x1C					b[2:0]		i	U[6:0]							
F16_BCC_REG		0x1D					b[2:0]		i	S[7:1]							
F16_BCC	F16_B_S	0x1E					0 0		S[9:1]								
	F16_BEQ_S						0 1										
	F16_BNE_S						1 0										
	F16_BCC_S						1 1		i[2:0]		S[6:1]						
F16_BL		0x1F					S[12:2]										

7.1.6 Encoding Notation

This chapter explains the full encoding details along with the shortened form, represented by a set of characters, used in “[Instruction Set Details](#)” on page 409. [Table 7-16](#) on page 314 lists the syntax conventions.

All fields that correspond to a 32-bit instruction word for a particular format are shown. Fields that have pre-defined values assigned to them are illustrated, and fields that are encoded by the assembler are represented as letters.



Note

Address offset bits that are always zero, due to address alignment, are omitted from the encoding of the offset to save space. For example, `LD_S R0, [GP, s11]` has an 11-bit offset, but only the upper 9 bits are encoded. The least-significant two offset bits are always 00 because all word accesses must be word aligned.

[Table 7-5](#) on page 306 and [Table 7-6](#) on page 307 list the notation used for the encoding.

Table 7-5 Key for 32-bit Addressing Modes and Encoding Conventions

Encoding Character	Encoding Field	Syntax
I	I[4:0]	instruction major opcode
i	i[n:0]	instruction sub opcode
A	A[5:0]	destination register
b	B[2:0]	lower bits source/destination register
B	B[5:3]	upper bits source/destination register
C	C[5:0]	source/destination register
Q	Q[4:0]	condition code
u	U[n:0]	unsigned immediate (number is bitfield size)
s	S[n:0]	lower bits signed immediate (number is bitfield size)
S	S[m:n+1]	upper bits signed immediate (number is bitfield size)
T	S[24:21]	upper bits signed immediate (branch unconditionally far)
w	W[5:0]	signed immediate 6-bit store data value
P	P[1:0]	operand format
M	M	conditional instruction operand mode
N	N	<.d> delay slot mode
F	F	Flag Setting
R	R	Reserved

Table 7-5 Key for 32-bit Addressing Modes and Encoding Conventions (Continued)

Encoding Character	Encoding Field	Syntax
D	Di	<.di> direct data cache bypass; this is not applicable to the ARC SEM processors
a	a[1:0]	<.aa> address write-back mode
Z	Z[1:0]	<.zz> data size
X	X	<.x> sign extend
K	K	<.k> store constant operand mode

Table 7-6 Key for 16-bit Addressing Modes and Encoding Conventions

Encoding Character	Encoding Field	Syntax
I	I[4:0]	instruction major opcode
i	i[n:0]	instruction sub-opcode
a	a[2:0]	source/destination register (r0-3,r12-15)
b	b[2:0]	source/destination register (r0-3,r12-15)
c	c[2:0]	source/destination register (r0-3,r12-15)
h	h[2:0]	source/destination register low (r0-r31 excluding r29 and r30)
H	h[4:3]	source/destination register high (r0-r31 excluding r29 and r30)
g	g[2:0]	destination register low (r0-r31 excluding r29 and r30)
G	G[4:3]	destination register high (r0-r31 excluding r29 and r30)
u	u[n:0]	unsigned immediate (number is bitfield size)
s	s[n:0]	signed immediate (number is bitfield size)

7.1.7 Long Immediate Source Operands and Null Destination Operands

Any 6-bit source register field can indicate that long immediate data is used in place of a register value. This is achieved by using the special register number 62 to indicate a long immediate. This can be used multiple times in an instruction. When a source register field is 62, an explicit 32-bit long immediate value follows the instruction word.

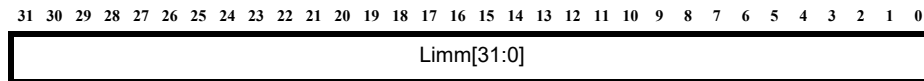
When a destination register field contains 62, the result of the instruction is discarded and no destination register is updated. Any status flag updates still occur according to the set-flags directive (.F) or if flag setting is implicit in the instruction.

If the long immediate indicator is used in both a source and the destination operand, a long immediate value is used as the source operand and the result is discarded as expected.

If the long immediate indicator is used in both source operands, the same long immediate value is used as the value for both source operands. This common long immediate source operand is not duplicated in memory; it is encoded only once and immediately after the instruction as usual.

For a detailed description of the layout in memory for instructions with long immediate data, see to the section on [“Instruction Layout in Memory”](#) on page 126.

Figure 7-1 Long Immediate source data value



Operand syntax:

limm - when used as a source operand (limm value is stored in memory after the instruction)

0 - when used as a destination operand (no value is stored in memory)

7.1.8 Condition Code Tests

[Table 7-7](#) lists the codes used for condition code tests.

Table 7-7 Condition Codes

Code Q field	Mnemonic	Condition	Test
0x00	AL, RA	Always	1
0x01	EQ, Z	Zero (Note that EQ and Z are equivalent encodings for condition codes)	Z
0x02	NE, NZ	Non-Zero	/Z
0x03	PL, P	Positive	/N
0x04	MI, N	Negative	N
0x05	CS, C, LO	Carry set, lower than (unsigned)	C
0x06	CC, NC, HS	Carry clear, higher or same (unsigned)	/C
0x07	VS, V	Over-flow set	V
0x08	VC, NV	Over-flow clear	/V
0x09	GT	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)
0x0A	GE	Greater than or equal to (signed)	(N and V) or (/N and /V)
0x0B	LT	Less than (signed)	(N and /V) or (/N and V)
0x0C	LE	Less than or equal to (signed)	Z or (N and /V) or (/N and V)

Table 7-7 Condition Codes (Continued)

Code Q field	Mnemonic	Condition	Test
0x0D	HI	Higher than (unsigned)	/C and /Z
0x0E	LS	Lower than or same (unsigned)	C or Z
0x0F	PNZ	Positive non-zero	/N and /Z

The remaining 16 condition codes (10-1F) are available for extension and are used to do the following:

- provide additional tests on the internal condition flags or
- test extension status flags from external sources or
- test a combination external and internal flags

If an extension condition code is used which is not implemented, an [Illegal Instruction](#) exception is raised.

**Note**

PNZ does not have an inverse condition.

7.1.9 Encoding Branch and Jump Delay Slot Modes

[Table 7-8](#) lists the codes used for delay slot modes on Branch and Jump instructions.

Table 7-8 Delay Slot Modes

N Bit	Mode	Operation
0		Only execute the next instruction when not jumping (default)
1	D	Always execute the next instruction

7.1.10 Encoding Static Branch Predictions

Some branch instructions encode a static prediction of their most likely outcome. This static prediction is encoded as a combination of an opcode bit and the sign of the relative branch displacement. This encoding allows such branches to have a default static prediction that maximizes the prediction accuracy according to the normal expectation that backwards going branches are more likely to be taken than non-taken. Thus, the default prediction is Backwards Taken, Forwards Not Taken, or BTFN. There is an alternative encoding of every BRcc, BBIT0, and BBIT1 instruction that inverts the default prediction. These branch instructions therefore have two encodings that can be selected by the assembler or compiler according to the expected outcome for each individual branch instruction. These encodings are identical except for bit 3 of the 32-bit instruction.

7.1.10.1 Assembly Syntax for Static Branch Predictions

[Table 7-9](#) lists the interpretation of the <.T> assembly syntax when encoding static branch predictions within the Y bit of a BRcc or BBITn instruction, according to the sign of the branch displacement.

Table 7-9 Encoding Static Branch Predictions for BRcc, BBIT0, and BBIT1

Instruction Grouping	<.T> Syntax	Prediction	Sign of Displacement	Prediction Mode	Y bit
BRcc	.nt	Not taken	Positive	BTFN	0
			Negative	FTBN	1
	.t	Taken	Positive	FTBN	1
			Negative	BTFN	0
BBIT0, BBIT1	.nt	Not taken	Positive	BTFN	1
			Negative	FTBN	0
	.t	Taken	Positive	FTBN	0
			Negative	BTFN	1

Compilers and assembler programmers must use the <.T> syntax to convey the predicted outcome of each BRcc or BBITn instruction to the assembler. The assembler uses the [Table 7-9](#) to set the Y bit in the instruction to the correct value, depending on the required prediction and the sign of the branch displacement.

The default static prediction, in the absence of any <.T> syntax, is always BTFN. Therefore, a BRcc instruction always has the Y bit set to 0 by default, whereas a BBITn instruction always has the Y bit set to 1 by default. [Table 7-9](#) lists these cases in bold font.

7.1.11 Encoding Load / Store Address Write-back Modes

[Table 7-10](#) lists the codes used for address write-back modes in Load and Store instructions.

Table 7-10 Address Write-Back Modes

AA bits	Address Mode	Memory Address Used	Register Value Write-back
00	No write-back	Reg + offset	No write-back
01	.A or AW	Reg + offset	Reg + offset Register updated pre memory transaction.
10	.AB	Reg	Reg + offset Register updated post memory transaction.

Table 7-10 Address Write-Back Modes (Continued)

AA bits	Address Mode	Memory Address Used	Register Value Write-back
11	.AS Scaled, no write-back	Reg + (offset << scaling_shift) Note that using the scaled address mode with 8-bit data size (LDB.AS or STB.AS) has undefined behavior and must not be used. scaling_shift is based on the data size mode, ZZ. For more information, see Table 7-13 scaling_shift is defined in Table 7-11 .	No write-back

Table 7-11 Scaling Shift

ZZ Code	ZZ Suffix	scaling_shift
00		2
01	B	undefined
10	H	1
11	D	2

If the destination of a Load instruction is the same as a source register, and the addressing mode defines an update to the source register, the value written to that register is the value returned from memory.

7.1.12 Encoding Load / Store Cache Bypass Mode

[Table 7-12](#) lists the codes used for direct to memory bypass modes in Load and Store instructions.



Note

This section is not applicable to the ARC SEM processors.

Table 7-12 Cache Bypass Modes

Di bit	Di Suffix	Access Mode
0		Default access to memory. Cached data memory access (<i>default, if no <.di> field syntax</i>)
1	DI	Direct to memory, bypassing data-cache (if available). Non-cached data memory access

7.1.13 Encoding Load / Store Data Sizes

[Table 7-13](#) lists the codes used for data size modes in Load and Store instructions.

Table 7-13 Load Store Data Sizes

ZZ Code	ZZ Suffix	Access Mode
00		Word (32-bit), default behavior
01	B	Byte
10	H	Half-word (16-bit)
11	D	Raises an Illegal Instruction exception if LL64_OPTION is disabled, otherwise indicates double-word data.

7.1.14 Encoding Load Data Extension Modes

[Table 7-14](#) lists the codes used to encode extension semantics in Load instructions.

Table 7-14 Load Data Extend Mode

X bit	X Suffix	Extension Semantics
0		If size is not Word(32-bit, ZZ = 00) then data is zero extended.
1	X	If size is not Word (32-bit, ZZ = 00) then data is sign extended

7.1.15 Encoding Store Data Source Mode

[Table 7-15](#) lists the field values used to encode the source of store data for certain Store instructions.

There is no requirement to specify an instruction suffix to differentiate between the two operand sources because this difference can be determined from the syntax of the store data operand.

Table 7-15 Store Data Source Mode

K bit	K Suffix	Store Data Source
0		The 'c' operand field (Bits 6 through 11) specifies the address of the core register from which the Store Data is sourced.
1		The 'c' operand field contains a signed 6-bit literal value to be stored.

7.1.16 Use of Reserved Encodings

In a given format, one or more bits of an encoding can be marked as *Reserved*. In some formats, an entire field may be reserved, such as when a register field is present in a given format but is not used in the particular opcode (such as a MOV in format 0x04, which does not use source 1).

The presence of reserved bits has the following effect:

- The processor ignores the reserved bits and does not generate an exception on an instruction based on the value assigned to reserved bits. The functionality of the instruction is not affected by the value assigned to reserved bits.

- The reserved bits must be set to 0 while encoding instructions.

7.1.17 Use of Illegal Encodings

The following are two major categories of illegal encodings:

- Unused field encodings, all of which are reserved for future use
- Illegal combinations of fields, all of which are specified explicitly in this document

7.1.17.1 Unused Field Encodings

Each field within an instruction encoding is able to encode a set of values, some of which may be unused. For example, within most major formats there are opcodes that are not used to encode any particular instruction. All such field values are reserved for future expansion.

There may also be operand or mode fields that have unused encodings. For example, the 2-bit data size field `<.zz>` in Load and Store instructions does not have a defined use for the encoding 11 when `LL64_OPTION` is disabled. This field represents an unused field encoding.

Any use of such an unused field encoding is *Illegal*. Any attempt to execute an instruction containing an unused field encoding raises an [Illegal Instruction](#) exception.

7.1.17.2 Illegal Combinations of Fields

Fields are normally orthogonal, but certain combinations or values between 2 or more fields create an instruction whose behavior either does not have any meaning or cannot be realized. For example, a LD instruction with the sign-extension field `<.x>` set to 1 is not meaningful, and is therefore considered to be an illegal combination of the `<.x>` field and the operation code.

Illegal combinations of fields are specific to each instruction format, and are therefore defined in the sections of this document that define the formats.

Any use of an illegal combination of fields raises an [Illegal Instruction](#) exception.

7.2 Instruction Syntax Conventions

The mnemonics of instructions encoded in 16-bit formats normally carry a “_S” suffix to distinguish them from 32-bit formats, and where necessary a 32-bit format may carry a “_L” suffix.

`op` implies 32-bit instruction

`op_L` indicates 32-bit instruction

`op_S` indicates 16-bit instruction

Where the operator format (DOP, SOP and ZOP) is relevant to the syntax or encoding of the instruction.

If no suffix is used on the instruction, the implied format is a 32-bit format. Instructions encoded in 16-bit formats have a reduced range of source and target core registers unless indicated otherwise. For an alphabetic list of instructions, see [Table 10-1](#) on page 410. The following notation is used to describe the syntax of instructions.

Table 7-16 Instruction Syntax Convention

a	Destination register (reduced range for 16-bit instruction.)	
b	Source operand 1 (reduced range for 16-bit instruction) or destination register.	
c	Source operand 2 (reduced range for 16-bit instruction) or destination register.	
h, g	Full register range for 16-bit instructions; General Purpose register, capable of addressing registers r0-r28 and r31 in some of the 16-bit encoded instructions. When these operands encoded value is 30, the instruction indicates long immediate data rather than r30. And similarly, operand encoded value of 29 is reserved.	
A	A 64-bit destination register.	
B	A 64-bit source operand register.	
C	A 64-bit source operand register.	
ACC	Accumulator. In a 64-bit mode, the accumulator comprises a register pair, ACCH and ACCL.	
cc	Condition code	
<.cc>	Optional condition code	
IV	Invalid flag for floating-point unit operations	
DZ	Divide-by-zero flag for floating-point unit operations	
OF	Overflow flag for floating-point unit operations	
UF	Underflow flag for floating-point unit operations	
IX	Inexact flag for floating-point unit operations	
<.f>	Optional set flags	
<.aa>	Optional address write-back	
<.d>	Optional delay slot mode	
<.di>	Optional direct data cache bypass; This is not applicable to ARC SEM.	
<.x>	Optional sign extend	
<.zz>	Optional data size	
u	Unsigned immediate, number indicates field size	
s or t	Signed immediate, number indicates field size	
limm	Long immediate	
.T	Compilers and assembler programmers must use the <.T> syntax to convey the predicted outcome of each BRcc or BBITn instruction to the assembler.	

7.3 Status flags and conditions

The ARCV2 ISA is a flag-based architecture in which instructions may set one or more status flags as a side-effect of the operation being performed. A flag can indicate, for example, that the result is zero. Conditional instructions can check the flags. For example, a branch instruction may take a branch if the Zero flag is set.

There is a single set of status flags Z,N,C,V located in the [Status Register, STATUS32](#).

The ARCV2-based processor has an extensive instruction set, most of which can be either carried out conditionally, or set the flags, or both. However, instructions using short immediate operands cannot also include a condition-code test.

7.3.1 Flag Setting

Instructions encoded in 32-bit formats, which are able to set the flags, normally update the status flags only if the “set-flags directive” (.F) bit in the instruction format is set to 1. For some instructions, the primary result is the flags themselves, rather than an update of a general purpose register. Such instructions include CMP, RCMP, BTST and TST, for example. In the encoding of these instructions the set-flags bit is either reserved or always 1, and flag updates are unconditional.

Instructions encoded in 16-bit formats do not offer the option to set flags. However, flag updates are performed implicitly by a small number of 16-bit encoded instructions where the flag update is the primary result. This includes BTST_S, CMP_S, and TST_S.

7.3.2 Status Flags Notation

Each flag-setting instruction has specific rules governing how it may update the status flags, and how its behavior may be dependent on the value of the flags. These rules are explained for each instruction in the section entitled [Instruction Set Details](#). The following notation is used to describe updates to the status flags:

Z	= Set if result is zero
N	= Set if most significant bit of result is set
C	= Set if carry is generated
V	= Set if overflow is generated

The following conventions are used to identify whether each operation has an effect on each of the status flags:

•	= Set according to the result of the operation
	= Not affected by the operation
0	= Bit cleared after the operation
1	= Bit set after the operation

7.4 Arithmetic and Logical Instructions

These instructions perform either dyadic or monadic operators, and take one of the following forms:

$a \leftarrow b \text{ op } c$

$a \leftarrow \text{op } b$

A dyadic instruction applies the operator (op) to the source operands (b and c) and assigns the result to the destination operand (a). The ordering of the operands is important for non-commutative operations, such as, SUB, SBC, BIC, ADD1/2/3, and SUB1/2/3.

A monadic instructions applies the operator (op) to the source operand (b) and assigns the result to the destination operand (a).

All arithmetic and logical instructions can be conditional, or set the flags, or both.

The normal rules concerning the use of r62 to indicate long-immediate source data and/or a null result apply to these instructions (see “[Long Immediate Source Operands and Null Destination Operands](#)” on page 307).

7.4.1 Integer Adder Operations

Table 7-17 summarizes the instructions that perform integer addition, subtraction or comparison operations. These instructions are always supported in all ARCV2 processors.

Table 7-17 Integer addition, subtraction and comparison operations

Instruction	Operation	Description
ADD	$a \leftarrow b + c$	add
ADC	$a \leftarrow b + c + C$	add with carry
SUB	$a \leftarrow b - c$	subtract
SBC	$a \leftarrow (b - c) - C$	subtract with carry
CMP	$b - c$	compare
RCMP	$c - b$	reverse compare
RSUB	$a \leftarrow c - b$	reverse subtract
ADD1	$a \leftarrow b + (c \ll 1)$	add with left shift by 1
ADD2	$a \leftarrow b + (c \ll 2)$	add with left shift by 2
ADD3	$a \leftarrow b + (c \ll 3)$	add with left shift by 3
SUB1	$a \leftarrow b - (c \ll 1)$	subtract with left shift by 1
SUB2	$a \leftarrow b - (c \ll 2)$	subtract with left shift by 2
SUB3	$a \leftarrow b - (c \ll 3)$	subtract with left shift by 3

7.4.1.1 Add Instructions

The addition instructions perform 2's complement integer addition. There is a version of the ADD instruction (ADC) that includes the Carry flag into the sum, to support multi-length addition. There are versions of the ADD instruction that pre-shift the second source operand left by 1, 2 or 3 places before the addition (ADD1, ADD2 and ADD3). There are a number of 16-bit redundant formats for the add instructions, to assist with code density.

7.4.1.2 Subtract Instructions

The subtract instructions perform 2's complement integer subtraction. There is a version of the SUB instruction (SBC) that includes a borrow bit, obtained by inverting the Carry flag, to support multi-length arithmetic.

There are versions of the SUB instruction that pre-shift the second source operand left by 1, 2 or 3 places before the addition (SUB1, SUB2 and SUB3). There are a number of 16-bit redundant formats for the subtract instructions, to assist with code density.

There is a reverse subtraction instruction (RSUB), due to the asymmetry of some source operand formats and the non-commutativity of the subtraction operator.

7.4.1.3 Comparison Instructions

The compare instructions perform 2's complement integer subtraction with the sole purpose of setting the status flags Z, N, C and V, that is, the difference between the source operands is not assigned to a destination register. There is a 16-bit redundant format for the compare instruction, to assist with code density.

There is also a reverse comparison instruction (RCMP), due to the asymmetry of some source operand formats and the non-commutativity of the comparison operator.

7.4.2 Move Operations

Table 7-17 summarizes the instructions that perform move operations, including those that perform standard integer promotions on signed and unsigned byte and half-word quantities. These instructions are always supported in all ARCV2 processors.

Table 7-18 Move operations

Instruction	Operation	Description
MOV	$a \leftarrow b$	Move source to destination
EXTB	$a \leftarrow \text{extb}(b)$	Zero-extend byte to word
EXTH	$a \leftarrow \text{exth}(b)$	Zero-extend half to word
SEXB	$a \leftarrow \text{sexb}(b)$	Sign-extend byte to word
SEXH SEXW	$a \leftarrow \text{sexh}(b)$	Sign-extend half to word

7.4.2.1 Move Instructions

The primary function of the MOV instruction is to copy its source operand to its destination register. In addition, certain 32-bit encodings of MOV may be executed conditionally, and may optionally set the Z and

N status flags. Updates to the destination register can be avoided, in the normal way, by using a null destination operand (r62). Several 16-bit versions of MOV_S allow a restricted set of commonly-used operands to be referenced in a compact encoding.

Full details of all available formats and encodings are presented in the detailed description of MOV in Chapter 10, “Instruction Set Details”.

7.4.2.2 Zero-extension Instructions

The zero-extension instructions EXTB and EXTH convert byte and half-word operand values to full register width values respectively. For example, if a function argument is passed as a value of type `unsigned char`, it must be promoted to `int` before it can be added to another `int` value. This can be achieved by applying the EXTB instruction to the register containing the argument. Note, there is no requirement to explicitly extend values loaded from memory, as the LDB and LDH instructions perform implicit zero-extension of the byte or half-word value read from memory before assigning it to the destination register.

Full details of all available formats and encodings are presented in the detailed description of EXTB and EXTH in Chapter 10, “Instruction Set Details”.

7.4.2.3 Sign-extension Instructions

The sign-extension instructions SEXB and SEXH convert byte and half-word operand values to full register width values respectively. For example, if a function argument is passed as a value of type `char`, it must be promoted to `int` before it can be added to another `int` value. This can be achieved by applying the SEXB instruction to the register containing the argument. Note, there is no requirement to explicitly sign-extend values loaded from memory, as the LDB.X and LDH.X instructions perform implicit sign-extension of the byte or half-word value read from memory before assigning it to the destination register.

Full details of all available formats and encodings are presented in the detailed description of SEXB and SEXH SEXW in Chapter 10, “Instruction Set Details”.

7.4.3 Bit-wise Logical Operations

Table 7-19 summarizes the group of instructions that perform bit-wise logical operations. These instructions are always supported in all ARCV2 processors.

Table 7-19 Bit-wise logical operations

Instruction	Operation	Description
AND	$a \leftarrow b \text{ and } c$	logical bit-wise AND
OR	$a \leftarrow b \text{ or } c$	logical bit-wise OR
BIC	$a \leftarrow b \text{ and not } c$	logical bit-wise AND with invert
XOR	$a \leftarrow b \text{ exclusive-or } c$	logical bit-wise exclusive-OR
TST	$b \text{ and } c$	bit-wise test, updating flags only
NOT	$a \leftarrow \sim b$	bit-wise negation

7.4.4 Bit Operations and Mask Operations

Table 7-20 Bit-mask operations

Instruction	Operation	Description
BSET	$a \leftarrow b \text{ or } (1 \ll c)$	bit set
BCLR	$a \leftarrow b \text{ and not } (1 \ll c)$	bit clear
BTST	$b \text{ and } (1 \ll c)$	bit test
BXOR	$a \leftarrow b \text{ xor } (1 \ll c)$	bit xor
BMSK	$a \leftarrow b \text{ and } ((1 \ll (c+1)) - 1)$	bit mask
BMSKN	$a \leftarrow b \text{ and } \sim((1 \ll (c+1)) - 1)$	bit mask inverted

7.4.5 Shift and Rotate Operations

Table 7-21 Single Bit Shift and Rotate Operations

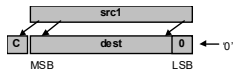
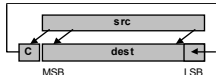
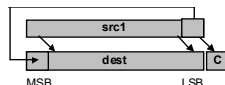
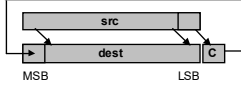
Instruction	Operation	Description
ASL		Arithmetic shift left by one
RLC		Rotate left through carry
ASR		Arithmetic shift right by one
LSR		Logical shift right by one
ROR		Rotate right

Table 7-21 Single Bit Shift and Rotate Operations

Instruction	Operation	Description
RRC		Rotate right through carry

7.4.5.1 Multi-bit Shift Operations

The multi-bit shifter provides a number of instructions that allow any operand to be shifted or rotated, left or right, by up to 32 positions, the result being available for write-back to any core register. [Table 7-21](#) on page 319 lists the single bit shift instructions provided as single operand instructions.

Table 7-22 Multi-bit Shift Operations

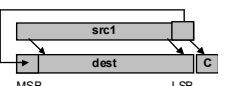
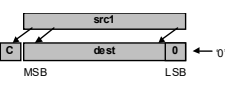
Instruction	Operation	Description
ASR multiple		Multiple arithmetic shift right, sign filled
LSR multiple		Multiple logical shift right, zero filled
ROR multiple		Multiple rotate right
ASL Multiple		Multiple arithmetic shift left, zero filled
XBFU	$a \leftarrow (b \gg c[4:0]) \text{ and } ((1 \ll (c[9:5] + 1)) - 1)$	Extract unsigned bit-field

Table 7-23 Shift-assist operations

Instruction	Operation	Description
LSL16	$a \leftarrow b \text{ lsl } 16$	Logical shift left 16 places
LSR16	$a \leftarrow b \text{ lsr } 16$	Logical shift right 16 places

Table 7-23 Shift-assist operations (Continued)

Instruction	Operation	Description
ASR16	$a \leftarrow b \text{ asr } 16$	Arithmetic shift right 16 places
ASR8	$a \leftarrow b \text{ asr } 8$	Arithmetic shift right 8 places
LSR8	$a \leftarrow b \text{ lsr } 8$	Logical shift right 8 places
LSL8	$a \leftarrow b \text{ lsl } 8$	Logical shift left 8 places
ROL8	$a \leftarrow b \text{ rol } 8$	Rotate left 8 places
ROR8	$a \leftarrow b \text{ ror } 8$	Rotate right 8 places

7.4.6 Selection Operations

Table 7-24 Selection operations

Instruction	Operation	Description
MIN	$a \leftarrow \min(b, c)$	Select smallest signed integer operand
MAX	$a \leftarrow \max(b, c)$	Select largest signed integer operand
ABS	$a \leftarrow \text{abs}(b)$	Select absolute value of b

7.4.7 Byte-swapping Operations

Table 7-25 Byte-swapping operations

Instruction	Operation	Description
SWAP	$b \leftarrow \text{swap}(c)$	Swap words
SWAPE	$b \leftarrow \text{swap\_endian}(c)$	Swap endianness

7.4.8 Bit-scanning Operations

Table 7-26 Bit-scanning operations

Instruction	Operation	Description
NORM	$b \leftarrow \text{norm}(c)$	Normalize
NORMH NORMW	$b \leftarrow \text{norm}(c)$	Normalize word
FFS	$a \leftarrow b \text{ ffs}(c)$	Find first set
FLS	$a \leftarrow b \text{ fls}(c)$	Find last set

7.4.9 Relational Comparison Operations

Table 7-27 Relational Comparison Operations

SETcc variant	Operation	Description
SETEQ	$a \leftarrow (b == c)$	Set if equal
SETNE	$a \leftarrow (b != c)$	Set if not equal
SETLT	$a \leftarrow (b < c)$ signed	Set if less than
SETGE	$a \leftarrow (b \geq c)$ signed	Set if greater than or equal
SETLO	$a \leftarrow (b < c)$ unsigned	Set if lower than
SETHS	$a \leftarrow (b \geq c)$ unsigned	Set if higher or same
SETLE	$a \leftarrow (b \leq c)$ signed	Set if less than or equal
SETGT	$a \leftarrow (b > c)$ signed	Set if greater than

7.4.10 Integer Multiply, Multiply-accumulate, and Divide Operations

Table 7-28 Integer Multiply, MAC and Divide Operations

Instruction	Operation	Description
MPY, MPY_S	$a \leftarrow b * c$	32 x 32 signed integer multiply, returning lower 32 bits of product
MPYU	$a \leftarrow b * c$	32 x 32 unsigned integer multiply, returning lower 32 bits of product
MPYM MPYH	$a \leftarrow b * c$	32 x 32 signed integer multiply, returning upper 32 bits of product
MPYMU MPYHU	$a \leftarrow b * c$	32 x 32 unsigned integer multiply, returning upper 32 bits of product
MPYW, MPYW_S	$a \leftarrow b * c$	16 x 16 signed integer multiply, returning 32 bit product
MPYUW, MPYUW_S	$a \leftarrow b * c$	16 x 16 unsigned integer multiply, returning 32 bit product
MPYD	$a \leftarrow b * c$	32 x 32 signed integer multiply, returning 64-bit result
MPYDU	$a \leftarrow b * c$	32 x 32 unsigned integer multiply, returning 64-bit result
MAX	$acc \leftarrow acc + (b * c)$ $a \leftarrow acc\_lo$	32 x 32 signed integer multiply-accumulate, returning lower 32 bits of the accumulated results
MACU	$acc \leftarrow acc + (b * c)$ $a \leftarrow acc\_lo$	32 x 32 unsigned integer multiply-accumulate, returning lower 32 bits of the accumulated results
MACD	$acc \leftarrow acc + (b * c)$ $A \leftarrow acc$	32 x 32 signed integer multiply-accumulate, returning 64 bits of the accumulated results
MACDU	$acc \leftarrow acc + (b * c)$ $A \leftarrow acc$	32 x 32 unsigned integer multiply-accumulate, returning 64 bits of the accumulated results

Table 7-28 Integer Multiply, MAC and Divide Operations

Instruction	Operation	Description
DIV	$a \leftarrow b / c$	Signed integer division, returning a 32-bit quotient
DIVU	$a \leftarrow b / c$	Unsigned integer division, returning a 32-bit quotient
REM	$a \leftarrow b \% c$	Signed integer division, returning the 32-bit remainder
REMU	$a \leftarrow b \% c$	Unsigned integer division, returning the 32-bit remainder

7.4.11 Dual and Quad Integer Multiply / Accumulate Operations

Table 7-29 Dual and Quad Integer Multiply and MAC Operations

Instruction	Operation	Description
DMPYH	if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}	Dual 16x16 multiply
DMPYHU	if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}	Dual 16x16 multiply unsigned
DMACH	if (cc) { result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result; }	Dual 16x16 MAC
DMACHU	if (cc) { result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result; }	Dual 16x16 MAC unsigned

7.4.12 Integer Vector Operations

Table 7-30 Integer Vector ADD, SUB, MPY operations

Instruction	Operation	Description
VADD2H	if (cc) { a.h0 = b.h0 + c.h0; a.h1 = b.h1 + c.h1; }	Vector-add, dual 16-bit
VSUB2H	if (cc) { a.h0 = b.h0 - c.h0; a.h1 = b.h1 - c.h1; }	Vector-sub, dual 16-bit

Table 7-30 Integer Vector ADD, SUB, MPY operations

Instruction	Operation	Description
VADDSUB2H	if (cc) { a.h0 = b.h0 + c.h0; a.h1 = b.h1 - c.h1; }	Vector-add/sub, dual 16-bit
VSUBADD2H	if (cc) { a.h0 = b.h0 - c.h0; a.h1 = b.h1 + c.h1; }	Vector-sub/add, dual 16-bit
VMPY2H	if (cc) { acc.w0 = A.w0 = b.h0 * c.h0; acc.w1 = A.w1 = b.h1 * c.h1; }	Vector-multiply, dual 16x16
VMPY2HU	if (cc) { acc.w0 = A.w0 = b.h0 * c.h0; acc.w1 = A.w1 = b.h1 * c.h1; }	Unsigned vector-multiply, dual 16x16

7.5 Memory Access Instructions

These instructions include data transfer operations from and to memory locations, and data transformation instructions such as sign-extend and rotate instructions.

Table 7-31 Memory Access Instructions

Instruction	Operation	Description
EX	$b \leftarrow \text{mem}[c];$ $\text{mem}[c] \leftarrow b$	Atomic Exchange
LD	Load register-register	See section General Operations Register-register Format on page 368
LDI	$b \leftarrow \text{memory}[\text{LDI\_BASE} + (\text{src1} \ll 2)]$	Load Indexed
LLOCK	$b \leftarrow \text{mem}[c];$ $\text{LF} \leftarrow 1;$	Load locked
SCOND	if $\text{LF mem}[c] \nabla b$	Store conditional
LLOCKD	$B \leftarrow \text{mem}[C];$ $\text{LF} \leftarrow 1;$	Load locked on 64-bit data
SCONDD	if $\text{LF mem}[C] \nabla B$	Store conditional on 64-bit data

The transfer of data to and from memory is accomplished with the load and store commands (LD, ST). These instructions can write the result of the address computation back to the address source register, pre- or post-calculation. This operation is accomplished with the optional address write-back suffixes: .A or .AW (register updated pre-memory transaction), and .AB (register updated post memory transaction). Addresses are interpreted as byte addresses unless the scaled address mode is used, as indicated by the address suffix .AS. The scaled address mode does not write back the result of the address calculation to the address source register.

**Note**

Do not use the scaled address mode with 8-bit data size (LDB.AS or STB.AS) as the behavior is undefined.

If the offset is not required during a load or store, the value encoded is set to 0.

The size of the data for a load or a store is indicated by the use of standard suffixes to represent 8-bit Byte (B) data, 16-bit Half-word (H) data, and 64-bit Double-word (D) data. The absence of a suffix implies 32-bit Word data. Therefore, the following instructions are provided: Load Byte instruction (LDB), Load Half-word instruction (LDH), Store Byte instruction (STB), and Store Half-word instruction (STH). The mnemonics LD or ST with no size suffix indicate 32-bit Word data. Byte and half-word loads may be sign-extended to 32-bits by using the sign extend suffix: .X. As a configurable option, the following instructions are also provided: Load Double (LDD) and Store Double (STD).

**Note**

Do not use the sign extend suffix on the LD instruction with a 32-bit data size as the resulting behavior is undefined.

If a data-cache is available in the memory controller, the load and store instructions can bypass the use of that cache. When the suffix .DI is used, the cache is bypassed, and the data is loaded directly from or stored directly to the memory. These memory instructions are useful for shared data structures in main memory, for the use of memory-mapped I/O registers, or for bypassing the cache to stop the cache being updated and overwriting valuable data that has already been loaded into cache.

7.5.1 Load Instructions

Two syntaxes are available, depending on how the address is calculated: register-register and register-offset. The syntax for the load instruction is:

```
LD<zz><.x><.aa><.di> a,[b]           (uses ld a,[b,0])
LD<zz><.x><.aa><.di> a,[b,s9]
LD<zz><.x><.di>          a,[limm,s9]   (Redundant format, use ld a,[limm])
LD<zz><.x><.di>          a,[limm]      (= ld a,[limm,0])
LD<zz><.x><.aa><.di> a,[b,c]
LD<zz><.x><.aa><.di> a,[b,limm]
LD<zz><.x><.di>          a,[limm,c]
LD_S                   a, [b, c]      (reduced set of regs)
```

LDB_S	a, [b, c]	(reduced set of regs)
LDH_S	a, [b, c]	(reduced set of regs)
LD_S	c, [b, u7]	(u7 offset is 32-bit aligned, reduced set of regs)
LDB_S	c, [b, u5]	(reduced set of regs)
LDH_S<.x>	c, [b, u6]	(u6 offset is 16-bit aligned, reduced set of regs)
LD_S	b, [SP, u7]	(u7 offset is 32-bit aligned, reduced set of regs)
LDB_S	b, [SP, u7]	(u7 offset is 32-bit aligned, reduced set of regs)
LD_S	r0, [GP, s11]	(s11 offset is 32-bit aligned, reduced set of regs)
LDB_S	r0, [GP, s9]	(reduced set of regs)
LDH_S	r0, [GP, s10]	(s10 offset is 16-bit aligned, reduced set of regs)
LD_S	b, [PCL, u10]	(u10 offset is 32-bit aligned, reduced set of regs)
LD_S	r, [h, u5]	(r is one of r0-r3, u5 offset is 32-bit aligned, reduced set of regs)
LD_S.AS	a, [b, c]	(reduced set of regs)
LD_S	r1, [GP, s9]	(reduced set of regs)

7.5.2 Store Instructions

7.5.2.1 Store Register with Offset

The following is the syntax for the Store register (**ST ST STH STB**) with offset instruction:

ST<zz><.aa><.di>	c, [b]	(use st c, [b, 0])
ST<zz><.aa><.di>	c, [b, s9]	
ST<zz><.di>	c, [limm]	(= st c, [limm, 0])
ST<zz><.aa><.di>	limm, [b, s9]	
ST_S	b, [SP, u7]	(u7 offset is 32-bit aligned)
STB_S	b, [SP, u7]	(u7 offset is 32-bit aligned)
ST_S	c, [b, u7]	(u7 offset is 32-bit aligned)

STB_S	c, [b, u5]
STH_S	c, [b, u6] <i>(u6 offset is 16-bit aligned)</i>

7.5.3 Stack Pointer Operations

ARCV2-based processors provide stack pointer functionality through the use of the stack pointer core register (SP). Push and pop operations are provided through normal Load and Store operations in the 32-bit instruction set, and specific instructions in the 16-bit instruction set. The following is the instructions syntax for push operations on the stack:

ST.AW	c,[SP,-4]	<i>(Push c onto the stack)</i>
PUSH_S	b	<i>(Push b onto the stack)</i>
PUSH_S	BLINK	<i>(Push BLINK onto the stack)</i>

The following is the instructions syntax for pop ([POP_S](#)) operations on the stack:

LD.AB	a,[SP,+4]	<i>(Pop top item of stack to a)</i>
POP_S	b	<i>(Pop top item of stack to b)</i>
POP_S	BLINK	<i>(Pop top item of stack to BLINK)</i>

The following instructions are also available in 16-bit instruction format, for working with the stack:

LD_S, LDB_S, ST_S, STB_S, ADD_S, SUB_S, MOV_S, and CMP_S.

7.5.4 Atomic Memory Operations

An atomic exchange operation, EX, is provided in the ARCV2 architecture as a primitive for multiprocessor synchronization allowing for atomic manipulation of semaphores in memory.

The EX performs an atomic exchange between a memory address and a core register. This operation is atomic in that the memory system ensures that the memory read and memory write cannot be separated by interrupts or by memory accesses from another processor or I/O device.

Two forms of the EX instructions are provided: an uncached form (using the .DI directive) for synchronization between multiple processors, and a cached form for synchronization between processes on a single-processor system. The cached EX is completely handled by the core and does not depend on the AXI subsystem. The uncached EX relies on locked bus transactions that must be supported in the memory subsystem. Note that the locked bus transactions is supported in AXI3 but it is not supported in AXI4.

The ARCV2 architecture also provides special Load-locked and Store-conditional instructions, to enable the implementation of lock-free data structures. These instructions generate exclusive transactions (in AXI this means the command attribute AxLOCK is asserted). The LLOCK instruction loads a value from memory, while at the same time setting an architecturally-invisible lock flag and lock address register. The SCOND instruction atomically checks the status of the lock flag and lock address, and if the check succeeds this instruction writes the result to memory.

Any interrupt or exception clears the lock flag, and any external write to the lock location also clears the lock flag. The lock location is defined to be the 4-byte word location given by the lock address when the LL64_OPTION is disabled, or the 8-byte double-word location given by the lock address when the LL64_OPTION is enabled. All LLOCK, LLOCKD, SCOND, and SCONDD instructions must specify addresses that are aligned to the size of the lock location, otherwise a misalignment exception is raised irrespective of the value of the STATUS32.AD bit. Thus, LLOCK and SCOND addresses must be aligned to a 4-byte boundary, and LLOCKD and SCONDD addresses must be aligned to an 8-byte boundary. The success or failure of the SCOND/SCONDD instruction is returned to the processor via the Zero flag (STATUS32[Z]), which is set to 1 if these instructions succeed in writing to memory, and is cleared if the write to memory did not take place.

The instruction syntax for all atomic memory access instructions is:

```
op<.di> b,[c]
op<.di> b,[limm]
op<.di> b,[u6]
```

Where “op” may be EX, LLOCK, LLOCKDD, SCOND, or SCONDD, depending on whether each type of atomic instruction is configured.

7.5.5 Prefetch Instructions

The PREFETCH instruction is used to initiate a data cache load without writing to any core register.

The ISA provides three types of prefetch instruction, although some cores may not support all of them. See the relevant core-specific Databook for details:

- PREFETCH: Prefetch data with no intention to write.
- PREFETCHW: Prefetch data with intention to write, that is, the line is allocated in dirty/modified state in the data cache. This option cannot be configured in the ARC EM processor, and therefore this instruction is decoded as an illegal instruction in the ARC EM processor.
- PREALLOC: Allocates a cache line as a preparation for writing the entire line. The line is not fetched from the memory subsystem. This option cannot be configured in the ARC EM processor, and therefore this instruction is decoded as an illegal instruction in the ARC EM processor.

All prefetch-type instructions are encoded as load instructions with a null destination register (r62), in either of the two 32-bit encodings for loads.

The semantics of each load instruction is governed by four orthogonal bit fields within the instruction format:

- <zz> specifies data size: word (00), byte (01), half-word (10) and double-word (11)
- <aa> specifies address update/scaling mode: none (00), pre-incr (01), post-incr (10), scaled (11)
- <d> specifies cache bypass mode: no-bypass (0), bypass (1)
- <x> specifies sign-extension: no sign-extension (0), sign-extension (1)

The <zz> and <aa> fields are orthogonally available also to prefetch-type instructions, allowing data size and addressing mode to be specified.

The <x> and <d> fields specify which particular prefetch-type instruction is encoded, when the destination register is r62 (null destination).

Some combinations of values for <zz>, <aa>, <d>, and <x> are illegal, although this may also depend on whether the instruction is a load or a prefetch-type.

The PREFETCH, PREFETCHW and PREALLOC instructions are encoded using <d> and <x> bits, when the destination register is null (r62), as follows:

Table 7-32 PREFETCH Instruction Encodings

Class	<d>	<x>	Instruction
PREFETCH	0	0	PREFETCH
PREFETCH	0	1	PREALLOC; This instruction is not supported in the ARC EM processor and is decoded as an illegal instruction.
PREFETCH	1	0	PREFETCHW; This instruction is not supported in the ARC EM processor and is decoded as an illegal instruction.
PREFETCH	1	1	Reserved (currently deemed illegal)
Load	-	-	LD, LDB, LDH or LDD, depending on <zz>

The following combinations of load or prefetch-type modifier fields are illegal, and raise an Illegal Instruction exception.

- The <x> and <d> bits together select the prefetch-type opcode.
- The '-' character indicates a don't care case.
- The <aa> and <zz> fields operate identically for both prefetch and load instruction classes. Thus, prefetch-type instructions may have both data-size and address pre/post-increment or scaling, just as load instructions.

All illegal combinations of <d>, <x>, <aa>, and <zz> are shown in [Table 7-33](#). These combinations cover all load and prefetch-type instructions in 32-bit encodings.

Table 7-33 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions

Class	<d>	<x>	<aa>	<zz>	Instruction
PREFETCH	1	1	-	-	Reserved combination of PREFETCH opcode
Load	-	1	-	00	Cannot sign-extend from 32 bits
Load	-	1	-	11	Cannot sign-extend from 64bits
-	-	-	11	01	Cannot scale an address for byte data

7.6 Auxiliary Register Operations

Access to the auxiliary register set is accomplished with the special load register and store register instructions (LR and SR). The LR instruction reads a specified auxiliary register location and places its value into the destination general-purpose register. The SR instruction writes the value obtained from a general-purpose source register to a specified auxiliary register location. The AEX instruction performs a simultaneous LR and SR, effectively exchanging the contents of a general-purpose register and an auxiliary register location.

Access to the auxiliary registers is limited to only 32 bit (word). The LR and SR instructions may *not* be conditional, but the AEX instructions is permitted to be conditional.

Table 7-34 Auxiliary Register Operations

Instruction	Operation	Description
AEX	$\text{tmp} \leftarrow \text{aux.reg}[c]$ $\text{aux.reg}[c] \leftarrow b$ $b \leftarrow \text{tmp}$	Auxiliary register exchange with core register
LR	$b \leftarrow \text{aux.reg}[c]$	Load from auxiliary register
SR	$\text{aux.reg}[c] \leftarrow b$	store to auxiliary register

7.6.1 Load from Auxiliary Register

The load from auxiliary register instruction, LR, has one source and one destination register. The LR instruction is not a conditional instruction and uses the [General Operations Register-register Format](#) format on [page 368](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 369](#), and the [General Operations Register with Signed 12-bit Immediate](#) format on [page 369](#).

7.6.2 Store to Auxiliary Register

The store to auxiliary register instruction, SR, has two source registers only. The SR instruction is not a conditional instruction and uses the [General Operations Register-register Format](#) format on [page 368](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 369](#), and the [General Operations Register with Signed 12-bit Immediate](#) format on [page 369](#).

7.6.3 Auxiliary Register Exchange

The auxiliary register exchange instruction, AEX, has one source and one destination register. The AEX instruction may be a conditional instruction and uses the [General Operations Register-register Format](#) format on [page 368](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 369](#), the [General Operations Register with Signed 12-bit Immediate](#) format on [page 369](#), and the [General Operations Conditional Register with Unsigned 6-bit Immediate](#) format on [page 370](#).

7.7 Control Flow Instructions

These instructions include the Branch, jump, conditional, and interrupt-related operations.

Table 7-35 Control Flow Operations

Instruction	Operation	Description
J	$pc \leftarrow c$	Unconditional jump
Jcc	if (cc) $pc \leftarrow c$	Conditional jump
JL	blink $\leftarrow$ next_pc; $pc \leftarrow c$	Jump and link
JLcc	if (cc) { blink $\leftarrow$ next_pc; $pc \leftarrow c$ }	Jump and link conditionally
JLI_S	blink $\leftarrow$ next_pc; $pc \leftarrow JLI\_BASE + (u10 \ll 2);$	Jump and link indexed
EI_S	BTA $\leftarrow$ next_pc; STATUS32.ES $\leftarrow$ 1 $pc \leftarrow EI\_BASE + (u10 \ll 2);$	Execute indexed
B B_S	$pc \leftarrow pcl + offset$	Branch unconditionally
Bcc Bcc_S	if (cc) { $pc \leftarrow pcl + offset$ }	Branch conditionally
BLcc	if (cc) { blink $\leftarrow$ next_pc; $pc \leftarrow pcl + offset$ }	Branch-and-link conditionally
BRcc	if (cmp(b,c)) { $pc \leftarrow pcl + offset$ }	Compare-and-branch
BBIT0	if (b[c] == 0) { $pc \leftarrow pcl + offset$ }	Branch if bit is zero
BBIT1	if (b[c] == 1) { $pc \leftarrow pcl + offset$ }	Branch if bit is one
BI	$pc \leftarrow next\_pc + (c \ll 2)$	Branch indexed
BIH	$pc \leftarrow next\_pc + (c \ll 1)$	Branch indexed half-word
LP<cc>	LP_END $\leftarrow$ pcl + s13 LP_START $\leftarrow$ next_pc	Unconditional loop setup (16-bit aligned target address)

Table 7-35 Control Flow Operations

Instruction	Operation	Description
LP<cc>	<pre> if (!cc) { LP_END ← pcl + u7 LP_START ← next_pc } else { pc ← pcl + u7 } </pre>	Conditional loop setup (16-bit aligned target address)
DBNZ	<pre> if (src != 1) { PC ← PCL + offset } src = src - 1 </pre>	Decrement register and branch if resulting value is non-zero.
ENTER_S	Function prolog sequence	Performs a programmable sequence of operations to allocate a stack frame and save the callee-saved registers. This is not strictly a control-flow instruction, but it is grouped most naturally with LEAVE_E, which can perform a jump to BLINK.
LEAVE_S	Function epilog sequence	Performs a programmable sequence of operations to restore callee-saved registers, deallocate a stack frame, and return to the caller.

7.7.1 Branch Instructions

Because of the pipeline in the ARCV2-based processor, a branch instruction does not take effect immediately, but after a delay of one or more cycles, depending on the implementation. The execution of the immediately following instruction after the branch can be enabled to minimize the cost of each branch instruction. Such a following instruction is said to be in the *delay slot*. The modes for specifying the execution of the delay slot instruction are indicated by the optional .d field according to the [Table 7-36](#).

Table 7-36 Delay Slot Execution Modes

Mode	Operation
ND	Only execute the next instruction when not jumping (default)
D	Always execute the next instruction

Because the execution of the instruction that is in the delay slot is controlled by the delay slot mode, that instruction must never be the target of any branch or jump instruction.

[Table 7-7](#) on page 308 lists the condition codes that are available for conditional branch instructions.

7.7.1.1 Conditional Branch Instructions

Conditional Branch (Bcc) has a branch range of $\pm 1\text{MB}$, whereas unconditional branch (B) has larger range of $\pm 16\text{MB}$. The branch target address is 16-bit aligned.

The following is the syntax of the branch instruction:

```

Bcc<.d>  s21  (branch if condition is true)
B<.d>    s25  (unconditional branch far)
B_S      s10  (unconditional branch)
BEQ_S    s10
BNE_S    s10
BGT_S    s7
BGE_S    s7
BLT_S    s7
BLE_S    s7
BHI_S    s7
BHS_S    s7
BLO_S    s7
BLS_S    s7

```

7.7.1.2 Branch and Link Instructions

Conditional Branch and Link (BLcc) has a branch range of $\pm 1\text{MB}$, whereas unconditional Branch-and-Link (BL) has larger range of $\pm 16\text{MB}$. The target address must be 32-bit aligned.

The BLINK register (see [Link Registers, ILINK \(r29\), BLINK \(r31\)](#)) is updated by the branch and link instruction to provide a link back to the position following the branch-and-link instruction.

Returning from a branch-and-link is accomplished by jumping to the contents of the BLINK register, using the Jcc [BLINK] instruction (see [Jcc](#) on page 596).

The following is the syntax of the branch and link instructions:

```

BLcc<.d> s21  (branch if condition is true)
BL<.d>    s25  (unconditional branch far)
BL_S      s13  (unconditional branch)

```

7.7.1.3 Branch On Compare or Bit Test

Branch on Compare (BRcc) and Branch on Bit Test (BBIT0, BBIT1) have a branch range of $\pm 256\text{B}$. The branch target address is 16-bit aligned.

The BRcc instruction is similar in execution to a normal compare instruction (CMP) with the addition that a branch occurs if the condition is met. No flags are updated and no ALU result is written back to the register file. A limited set of condition code tests are available for the BRcc instruction as shown in the following table. Table 7-37 lists the additional condition code tests that are available through the effect of reversing the operands.

Table 7-37 Branch on Compare/Test Mnemonics

Mnemonic	Condition
BREQ	Branch if b and c are equal
BRNE	Branch if b and c are not equal
BRLT	Branch if b is less than c
BRGE	Branch if b is greater than or equal to c
BRLO	Branch if b is lower than c
BRHS	Branch if b is higher than or same as c
BBIT0	Branch if bit c in register b is clear
BBIT1	Branch if bit c in register b is set

Table 7-38 Branch on Compare Pseudo Mnemonics, Register-Register

Mnemonic	Condition
BRGT b,c,s9	Branch if b is greater than c (encode as BRLT c,b,s9)
BRLE b,c,s9	Branch if b is less than or equal c (encode as BRGE c,b,s9)
BRHI b,c,s9	Branch if b is higher than c (encode as BRLO c,b,s9)
BRLS b,c,s9	Branch if b is lower than or same c (encode as BRHS c,b,s9)

Table 7-39 lists the assembly pseudo-instructions, for missing conditions using immediate data. These versions have a reduced immediate range of 0 to 62, instead of 0 to 63.

Table 7-39 Branch on Compare Pseudo Mnemonics, Register-Immediate

Mnemonic	Condition
BRGT b,u6,s9	Branch if b is greater than u6 (encode as BRGE b,u6+1,s9)
BRLE b,u6,s9	Branch if b is less than or equal u6 (encode as BRLT b,u6+1,s9)
BRHI b,u6,s9	Branch if b is higher than u6 (encode as BRHS b,u6+1,s9)
BRLS b,u6,s9	Branch if b is lower than or same u6 (encode as BRLO b,u6+1,s9)

7.7.2 Jump Instructions

Because of the pipeline in the ARCV2-based processor, the jump instruction does not take effect immediately, but after a one-cycle delay. The execution of the immediately-following instruction after the jump can be controlled. The following instruction is said to be in the *delay slot*. The modes for specifying the execution of the delay slot instruction are indicated by the optional .d field according to the [Table 7-40](#).

Table 7-40 Delay Slot Execution Modes

Mode	Operation
ND	Only execute the next instruction when <i>not</i> jumping (default)
D	Always execute the next instruction

Because the execution of the instruction that is in the delay slot is controlled by the delay slot mode, that instruction must never be the target of any branch or jump instruction.



Note

Jump instructions with delay slots are not permitted to use long immediate data as the jump target operand. Any attempt to execute such an instruction raises an Illegal Instruction Exception.

When source register BLINK is used with the Jump instruction, the register provides a return from branch-and-link or a return from jump-and-link (see [Link Registers, ILINK \(r29\), BLINK \(r31\)](#)).

7.7.3 Zero Overhead Loop Instruction

The ARCV2-based processor has the ability to perform loops without any delays being incurred by the count decrement or the end address comparison. Zero-delay loops are set up with the registers LP_START, LP_END, and LP_COUNT. LP_START and LP_END can be directly manipulated with the LR and SR instructions and LP_COUNT can be manipulated in the same way as registers in the core register set.

The special instruction LP is used to set up the LP_START and LP_END in a single instruction. The LP instruction is similar to the branch instruction. Loops can be conditionally entered. If the condition code test for the LP instruction returns *false*, a branch occurs to the address specified in the LP instruction. The branch target address is 16-bit aligned. If the condition code test is *true*, the address of the next instruction is loaded into LP_START register and the LP_END register is loaded by the address defined in the LP instruction.

The loop instruction, LP, has a special syntax that ignores the destination field, and only requires one source operand. The source operand is a 16-bit aligned target address value.

7.7.4 Return from Interrupt or Exception Instruction

The return from interrupt or exception instruction, RTIE, allows exit from interrupt and exception handlers, and to allow the processor to switch from kernel mode to user mode.

7.8 Special Instructions

This section details instructions that are related to Interrupts, auxiliary registers, and zero operand instructions.

The following is the list of special instructions

Table 7-41 Special Instructions

Instruction	Operation	Description
NOP	No operation	No operation
SLEEP	Put processor in sleep state	Sleep until interrupt or restart
SWI	Software interrupt	Raise EV_SWI exception
SETI	Set or restore interrupt enable and priority level	Set or restore interrupt enable and priority level
CLRI	Clear Interrupt Enable	Disable interrupts
BRK	Break (halt) processor	Stop the processor
TRAP_S	Trap to system call	raise an EV_Trap exception with parameter n
UNIMP_S	Unimplemented Instruction	Unimplemented Instruction
SYNC	Synchronize with memory	Synchronize with memory
NEXTTASK	Switch the task and register bank	Switch the task and register bank. Start execution from the next PC

Syntax for special instructions:

Some instructions require no source operands or destinations. The ARCV2 ISA supports these instructions using the form **op c** where the operand source c supplies information for the instruction. Zero operand instructions can set the flags.

NOP (encoded as MOV 0,0)
NOP_S (16-bit instruction form)
SLEEP u6
SLEEP c
CLRI u6
CLRI c
SETI u6
SETI c
SWI (Software interrupt, 32-bit format)
SWI_S (Software interrupt, 16-bit format)
BRK_S (Breakpoint halt instruction, 16-bit format)

BRK (Breakpoint halt instruction, 32-bit format)

TRAP_S u6

UNIMP_S

RTIE

SYNC

op<.f> c

op<.f> u6

op<.f> limm

7.8.1 Breakpoint Instruction

The breakpoint instruction is a single operand basecase instruction that halts the processor from performing any instructions beyond the breakpoint. After execution of a breakpoint instruction, an external debug host is required to restart the processor.

7.8.2 Sleep Instruction

The sleep state is entered when the processor executes the SLEEP instruction. The processor stays in the sleep state until an interrupt or restart occurs. Power consumption may be reduced during the sleep state, depending on the capabilities of each specific implementation. The SLEEP instruction is serializing, which means the SLEEP instruction completes and then flushes the pipeline.

7.8.3 Software Interrupt Instruction

The SWI and SWI_S instructions perform a similar function to the BRK and BRK_S instructions, except that they do not halt the processor. These instructions cause the processor to raise the EV_SWI exception without actually completing the execution of the SWI or SWI_S instruction. The effect is to stop the program before it executes the software interrupt instruction. This instruction design allows native debuggers to insert soft breakpoints and handle them on the same processor.

7.8.4 SETI

Globally enable interrupts while optionally setting a priority level. The following sample SETI instruction enables interrupts and restores the preempting interrupt priority value.

```
SETI R0
```

7.8.5 CLRI

CLRI always forces the STATUS32.IE bit to 0, disabling interrupts. The following sample CLRI instruction clears interrupts and captures the interrupt state in R0.

```
CLRI R0
```

7.8.6 Trap Instruction

The instruction, TRAP_S, raises an exception and calls any operating system in kernel mode. Traps can be raised from user or kernel mode. The TRAP_S instruction completes its update to the program counter before raising the EV_Trap exception.

7.8.7 Synchronization Instructions

The synchronize instruction, SYNC, waits until all data-based memory operations (LD, ST, cache fills) have completed.

7.9 APEX Extension Instructions

These operations are generally of the form $a \leftarrow b \text{ op } c$ where the destination (a) is replaced by the result of the operation (op) on the operand sources (b and c). All extension instructions can be conditional or set the flags or both.

7.9.1 Syntax for Generic Extension Instructions

If the destination register is set to an absolute value of 0, the result is discarded and the operation acts like a NOP instruction. A long immediate (limm) value can be used for either source operand 1 or source operand 2. The generic extension instruction format is:

```

op<.f>      a,b,c
op<.f>      a,b,u6
op<.f>      b,b,s12
op<.cc><.f>  b,b,c
op<.cc><.f>  b,b,u6
op<.f>      a,limm,c    (if b=limm)
op<.f>      a,limm,u6
op<.f>      0,limm,s12
op<.cc><.f>  0,limm,c
op<.cc><.f>  0,limm,u6
op<.f>      a,b,limm    (if c=limm)
op<.cc><.f>  b,b,limm
op<.f>      a,limm,limm (if b=c=limm)
op<.cc><.f>  0,limm,limm
op<.f>      0,b,c       (if a=0)
op<.f>      0,b,u6

```

op<.f>	0,limm,c	(if a=0, b=limm)
op<.f>	0,limm,u6	
op<.f>	0,b,limm	(if a=0, c=limm)
op<.f>	0,limm,limm	(if a=0, b=c=limm)
op_S	b,b,c	

7.9.2 Syntax for Single Operand Extension Instructions

Single source operand instructions are supported for extension instructions. The following is the syntax for single operand instruction:

op<.f>	b,c
op<.f>	b,u6
op<.f>	b,limm
op<.f>	0,c
op<.f>	0,u6
op<.f>	0,limm
op_S	b,c

7.9.3 Syntax for Zero Operand Extension Instructions

The following is the syntax for zero operand instruction:

op<.f>	c
op<.f>	u6
op<.f>	limm
op_S	

7.10 Floating-Point Instructions

Table 7-42 Floating-Point Unit Instructions, 0x06

Sub-opcode B field (6 bits)	Instruction	F Bit	Description
0x00	FSMUL	0	Single-precision floating-point multiply
0x01	FSADD	0	Single-precision floating-point addition
0x02	FSSUB	0	Single-precision floating-point subtraction
0x03	FSCMP	1	Single-precision floating-point comparison
0x04	FSCMPF	1	Single-precision floating-point comparison operation with IEEE 754 flag generation.
0x05	FSMADD	0	Single-precision floating-point fusedMultiplyAdd
0x06	FSMSUB	0	Single-precision floating-point fusedMultiplySubtract
0x07	FSDIV	0	Single-precision floating-point division
0x08	FCVT32	0	Convert between two 32-bit data formats
0x2F	FSSQRT	0	Single-precision floating-point square root
BCLR r1, r2, 0x1F	FSABS		Single-precision floating-point absolute operation
BXOR r1, r2, 0x1F	FSNEG		Single-precision floating-point negate operation
0x30	if(cc) {	F	Double-precision floating point multiply by D1, writing to D1
0x31	DMULH12	F	Double-precision floating point multiply by D2, writing to D1
0x32	DMULH21	F	Double-precision floating point multiply by D1, writing to D2
0x33	DMULH22	F	Double-precision floating point multiply by D2, writing to D2
0x34	DADDH11	F	Double-precision floating point add to D1, writing to D1
0x35	DADDH12	F	Double-precision floating point add to D2, writing to D1
0x36	DADDH21	F	Double-precision floating point add to D1, writing to D2

Table 7-42 Floating-Point Unit Instructions, 0x06

Sub-opcode B field (6 bits)	Instruction	F Bit	Description
0x37	DADDH22	F	Double-precision floating point add to D2, writing to D2
0x3C	DEXCL1	F	Double-precision floating point register D1 exchange
0x3D	DEXCL2	F	Double-precision floating point register D2 exchange
0x38	DSUBH11	F	Double-precision floating point subtract from D1, writing to D1
0x39	DSUBH12	F	Double-precision floating point subtract from D2, writing to D1
0x3A	DSUBH21	F	Double-precision floating point subtract from D1, writing to D2
0x3B	DSUBH22	F	Double-precision floating point subtract from D2, writing to D2

7.10.0.1 Syntax for the Single Precision Floating Point Instructions

The floating point instructions provide fast single precision IEEE754 format operations with limited support for rounding, denormalization handling and exceptions.

The syntax for the floating point instructions is:

op<.f>	a,b,c
opL<.f>	b,b,u6
op<.f>	c,b,s12
op<.cc><.f>	b,b,c
op<.cc><.f>	b,b,u6
op<.f>	a,limm,c
op<.f>	a,b,limm
op<.cc><.f>	b,b,limm
op<.f>	0,b,c
op<.f>	0,b,u6
op<.f>	0,b,limm

op<.cc><.f> 0,limm,c

7.10.0.2 Syntax for the Double Precision Floating Point Instructions

The floating point instructions provide fast double precision IEEE754 format operations with limited support for rounding, de-normalization handling and exceptions.

The syntax for the floating point instructions is:

op<.f>	a,b,c
opL<.f>	b,b,u6
op<.f>	c,b,s12
op<.cc><.f>	b,b,c
op<.cc><.f>	b,b,u6
op<.f>	a,limm,c
op<.f>	a,b,limm
op<.cc><.f>	b,b,limm
op<.f>	0,b,c
op<.f>	0,b,u6
op<.f>	0,b,limm
op<.cc><.f>	0,limm,c

7.11 DSP Instructions

The DSP instruction set summary defines the following classes of instructions:

- [Basic Saturating Arithmetic Operations](#)
- [Vector Unpacking Operations](#)
- [Vector ALU Operations](#)
- [Accumulator Operations](#)
- [Vector SIMD 16x16 MAC Operations](#)
- [Dual Inner Product 16x16 MAC Operations](#)
- [32x32 and 16x16 MAC Operations](#)
- [Dual 16x8 MAC Operations](#)
- [Complex Operations](#)
- [32x16 MAC Operations](#)

Table 7-43 Instruction Mnemonics

Mnemonic Prefixes	
V	SIMD-style vector operation
D	Inner product style vector operation
C	Complex operation
Mnemonic Suffixes	
2	Two-way vector operation
B	Eight-bit operands
H	16-bit operands
WH	32x16 bit operands
HB	16x8 bit operands
HWF	16-bit fractional operands; 32-bit write-back
D	32x32 with 64-bit write-back
F	Fractional operand
T	24-bit operands
R	Rounding
U	Unsigned
C	Complex conjugate

Table 7-43 Instruction Mnemonics

L	Least significant
M	Most significant

7.11.1 Basic Saturating Arithmetic Operations

Table 7-44 Basic Saturating Arithmetic Operations

Instruction	Type	Major	Minor	Description
SATF	SOP	0x05	0x1A	Saturate value based on flag
SATH	SOP	0x05	0x02	Saturate a 32-bit value to a 16-bit value
RNDH	SOP	0x05	0x03	Round and saturate a 32-bit value to a 16-bit value
ABSSH	SOP	0x05	0x04	Compute the absolute value of a 16-bit element and saturate the result
ABSS	SOP	0x05	0x05	Compute the absolute value of a 32-bit operand and saturate the result
NEGSH	SOP	0x05	0x06	Negate the lower 16 bits of a word and saturate the result
NEGS	SOP	0x05	0x07	Negate a 32-bit word and saturate the result
ADDS	DOP	0x05	0x06	32-bit addition, and result is saturated
SUBS	DOP	0x05	0x07	32-bit subtraction, and result is saturated
ASLS	DOP	0x05	0x0A	Perform an arithmetic shift left operation on an operand. The shift amount specified in the second operand. Store the saturated result in the destination
ASRS	DOP	0x05	0x0B	Perform an arithmetic shift right operation on an operand. The shift amount specified in the second operand. Store the saturated result in the destination

Table 7-44 Basic Saturating Arithmetic Operations

Instruction	Type	Major	Minor	Description
ASRSR	DOP	0x05	0x0C	Perform an arithmetic shift right operation on an operand. The shift amount specified in the second operand. Store the rounded saturated result in the destination
ADCS	DOP	0x05	0x26	32-bit add with carry, saturating the result
SBCS	DOP	0x05	0x27	32-bit subtract with borrow, saturating the result

7.11.2 Vector Unpacking Operations



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in “[Expansion of Literals in Vector Instructions](#)” on page 118. Short immediates are replicated for each vector lane.

Table 7-45 Vector Unpacking Operations

Instruction	Type	Major	Minor	F bit	Description
VEXT2BHLF	SOP	0x05	0x20	0	Compose a vector with two Q15 elements by expanding the lower two Q7 bytes of an operand.
VEXT2BHMF	SOP	0x05	0x21	0	Compose a vector with two Q15 elements created by expanding the higher two Q7 bytes of an operand.
VREP2HL	SOP	0x05	0x22	0	Replicate lower 16-bits into the upper 16-bits
VREP2HM	SOP	0x05	0x23	0	Replicate higher 16-bits to lower 16-bits
VEXT2BHL	SOP	0x05	0x24	0	Vector extend the lower two bytes to the higher two bytes
VEXT2BHM	SOP	0x05	0x25	0	Vector extend the higher two bytes to the lower two bytes

Table 7-45 Vector Unpacking Operations

Instruction	Type	Major	Minor	F bit	Description
VSEXT2BHL	SOP	0x05	0x26	0	Vector sign extend the lower two bytes to the higher two bytes
VSEXT2BHM	SOP	0x05	0x27	0	Vector sign extend the higher two bytes to the lower two bytes
VPACK2HL	DOP	0x05	0x29	0	Compose the destination operand from the lower 16-bits of the source operands.
VPACK2HM	DOP	0x05	0x29	1	Compose the destination operand from the higher 16-bits of the source operands.
VPACK2HBL	SOP	0x05	0x1C	0	Pack the LSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the lower 16 bits of the destination.
VPACK2HBLF	SOP	0x05	0x1D	0	Pack the MSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the lower 16 bits of the destination.
VPACK2HBM	SOP	0x05	0x1E	0	Pack the LSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the higher 16 bits of the destination.
VPACK2HBMF	SOP	0x05	0x1F	0	Pack the MSBs of a 2x16 bit vectors into a 2x8 bit vector. Result is stored in the higher 16 bits of the destination.
VALGN2H	DOP	0x05	0x0D	0	Two-way 16-bit vector align

7.11.3 Vector ALU Operations



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in [“Expansion of Literals in Vector Instructions”](#) on page 118. Short immediates are replicated for each vector lane.

Table 7-46 Vector ALU Operations

Instruction	Type	Major	Minor	F bit	Description
VABSS2H	SOP	0x05	0x29	0	Two way 16 bit vector saturating absolute
VNEG2H	SOP	0x05	0x2A	0	Two way 16 bit vector negative
VNEGS2H	SOP	0x05	0x2B	0	Two way 16 bit vector saturating negative
VNORM2H	SOP	0x05	0x2C	0	Two way 16 bit vector normalization
VADD2H	DOP	0x05	0x14	0	Two way vector add 16 bits
VADDS2H	DOP	0x05	0x14	1	Two way vector saturating add 16 bits
VSUB2H	DOP	0x05	0x15	0	Two way vector subtract 16 bits
VADDSUB2H	DOP	0x05	0x16	0	Two way vector add subtract
VADDSUBS2H	DOP	0x05	0x16	1	Two way vector saturating add subtract
VSUBADD2H	DOP	0x05	0x17	0	Two way vector subtract add
VSUBADDS2H	DOP	0x05	0x17	1	Two way vector saturating subtract add
VASL2H	DOP	0x05	0x21	0	Two way arithmetic shift left 16 bits
VASLS2H	DOP	0x05	0x21	1	Two way saturating arithmetic shift left 16 bits
VASR2H	DOP	0x05	0x22	0	Two way arithmetic shift right 16 bits
VASRS2H	DOP	0x05	0x22	1	Two way arithmetic saturating shift right 16 bits
VLSR2H	DOP	0x05	0x23	0	Two way vector logic shift right 16 bits
VASRSR2H	DOP	0x05	0x23	1	Two way arithmetic saturating and rounding shift right 16 bits
VMAX2H	DOP	0x05	0x24	1	Two way vector maximum 16 bits
VSUB4B	DOP	0x05	0x25	0	Four way subtract

Table 7-46 Vector ALU Operations

Instruction	Type	Major	Minor	F bit	Description
VMIN2H	DOP	0x05	0x25	1	Two way vector minimum 16 bits

7.11.4 Accumulator Operations

Table 7-47 Accumulator Operations

Instruction	Type	Major	Minor	F bit	Description
ASLACC	ZOP	0x05	0x00	0	Arithmetic shift left accumulator
ASLSACC	ZOP	0x05	0x01	0	Saturating arithmetic shift left accumulator
FLAGACC	ZOP	0x05	0x04	1	Copy accumulator status flags to STATUS32
GETACC	SOP	0x05	0x18	0	Read accumulator window
NORMACC	SOP	0x05	0x19	0	Normalize the accumulator
SETACC	DOP	0x05	0x0D	1	Set accumulator

7.11.5 Vector SIMD 16x16 MAC Operations



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in “[Expansion of Literals in Vector Instructions](#)” on page 118. Short immediates are replicated for each vector lane.

Table 7-48 Vector 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
VMSUB2HFR	DOP	0x06	0x3	0	Two-way vector 16x16 signed fractional multiply-subtract, saturating and rounding
VMSUB2HF	DOP	0x06	0x4	0	Two-way vector 16x16 signed fractional multiply-subtract, saturating

Table 7-48 Vector 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
VMAC2HNFR	DOP	0x06	0x11	0	Two way vector 16 bit saturating and rounding fractional multiplication and accumulation
VMSUB2HNFR	DOP	0x06	0x11	1	Two-way vector 16x16 signed fractional multiply-subtract, saturating and rounding, no fractional shift
VMPY2HF	DOP	0x05	0x1C	0	Two way vector signed 16 bit multiplication
VMPY2H	DOP	0x05	0x1C	1	Two way vector 16 bit saturating fractional multiplication
VMPY2HU	DOP	0x05	0x1D	0	Two way vector unsigned 16 bit multiplication
VMPY2HFR	DOP	0x05	0x1D	1	Two way vector 16 bit saturating and rounding fractional multiplication
VMAC2H	DOP	0x05	0x1E	0	Two way vector 16 bit signed integer multiplication and accumulation
VMAC2HF	DOP	0x05	0x1E	1	Two way vector 16 bit saturating fractional multiplication and accumulation
VMAC2HU	DOP	0x05	0x1F	0	Two way vector 16 bit unsigned integer multiplication and accumulation
VMAC2HFR	DOP	0x05	0x1F	1	Two way vector 16 bit saturating and rounding fractional multiplication and accumulation
VMPY2HWF	DOP	0x05	0x20	0	Two way vector 16 bit saturating fractional multiplication; two way 32 bit write back

7.11.6 Dual Inner Product 16x16 MAC Operations



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in “[Expansion of Literals in Vector Instructions](#)” on page 118. Short immediates are replicated for each vector lane.

Table 7-49 Dual 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
DMPYH	DOP	0x05	0x10	F	Dual vector 16 bit signed integer multiplication
DMPYHU	DOP	0x05	0x11	F	Dual vector 16 bit unsigned integer multiplication
DMACH	DOP	0x05	0x12	F	Dual vector 16 bit signed integer multiplication and accumulation
DMACHU	DOP	0x05	0x13	F	Dual vector 16 bit unsigned integer multiplication and accumulation
DMPYHWF	DOP	0x05	0x28	F	Dual vector 16 bit signed fractional multiplication and 32-bit write back
DMPYHF	DOP	0x05	0x2A	F	Dual vector 16 bit signed fractional multiplication
DMPYHFR	DOP	0x05	0x2B	F	Dual vector 16 bit signed fractional rounding multiplication
DMACHF	DOP	0x05	0x2C	F	Dual vector 16 bit signed fractional multiplication and accumulation
DMACHFR	DOP	0x05	0x2D	F	Dual vector 16 bit signed fractional rounding multiplication and accumulation

7.11.7 32x32 and 16x16 MAC Operations

Table 7-50 32x32 and 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MPY	DOP	0x04	0x1A	F	32 bit signed integer multiplication
MPYM MPYH	DOP	0x04	0x1B	F	32 bit signed integer multiplication
MPYMU MPYHU	DOP	0x04	0x1C	F	32 bit unsigned integer multiplication
MPYU	DOP	0x04	0x1D	F	32 bit unsigned integer multiplication
MPYW	DOP	0x04	0x1E	F	16 bit signed integer multiplication
MPYUW	DOP	0x04	0x1F	F	16 bit unsigned integer multiplication

Table 7-50 32x32 and 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MAC	DOP	0x05	0x0E	F	32 bit signed integer multiplication and accumulation
MACU	DOP	0x05	0x0F	F	32 bit unsigned integer multiplication and accumulation
MPYD	DOP	0x05	0x18	F	32 bit signed integer multiplication; 64-bit result
MPYDU	DOP	0x05	0x19	F	32 bit unsigned integer multiplication; 64-bit result
MACD	DOP	0x05	0x1A	F	32 bit signed integer multiplication and accumulation; 64-bit result
MACDU	DOP	0x05	0x1B	F	32 bit unsigned integer multiplication and accumulation; 64-bit result
MPYF	DOP	0x06	0x0A	F	32 bit signed fractional saturating multiplication
MPYFR	DOP	0x06	0x0B	F	32 bit signed fractional saturating an rounding multiplication
MACF	DOP	0x06	0x0C	F	32 bit signed fractional saturating multiplication and accumulation
MACFR	DOP	0x06	0x0D	F	32 bit signed fractional saturating an rounding multiplication and accumulation
MSUBF	DOP	0x06	0x0E	F	32 bit fractional saturating subtraction
MSUBFR	DOP	0x06	0x0F	F	32 bit fractional saturating and rounding subtraction
MPYDF	DOP	0x06	0x12	F	32 bit signed fractional saturating multiplication; 64-bit result
MACDF	DOP	0x06	0x13	F	32 bit signed fractional saturating multiplication and accumulation; 64-bit result
MSUBDF	DOP	0x06	0x15	F	32 bit signed fractional saturating subtraction; 64-bit result
MPYW_S	DOP	0x0F	0x09	F	16 bit signed integer multiplication

Table 7-50 32x32 and 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MPYUW_S	DOP	0x0F	0x0A	F	16 bit unsigned integer multiplication
MPY_S	DOP	0x0F	0x0C	F	32 bit signed integer multiplication

7.11.8 Dual 16x8 MAC Operations



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in “[Expansion of Literals in Vector Instructions](#)” on page 118. Short immediates are replicated for each vector lane.

Table 7-51 Dual 16x8 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
DMPYHBL	DOP	0x06	0x16	F	Dual vector 16 bitx8 bit signed integer multiplication; lower bytes
DMPYHBM	DOP	0x06	0x17	F	Dual vector 16 bitx8 bit signed integer multiplication; higher bytes
DMACHBL	DOP	0x06	0x18	F	Dual vector 16 bitx8 bit signed integer multiplication and accumulation; lower bytes
DMACHBM	DOP	0x06	0x19	F	Dual vector 16 bitx8 bit signed integer multiplication and accumulation; higher bytes

7.11.9 32x16 MAC Operations

Table 7-52 32x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MPYWHL	DOP	0x06	0x1C	F	32x16 bit signed multiplication, lower signed 16 bits
MACWHL	DOP	0x06	0x1D	F	32x16 bit signed multiplication and accumulation, lower signed 16 bits
MPYWHUL	DOP	0x06	0x1E	F	32x16 bit unsigned multiplication, lower signed 16 bits

Table 7-52 32x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MACWHUL	DOP	0x06	0x1F	F	32x16 bit unsigned multiplication and accumulation, lower signed 16 bits
MPYWHFM	DOP	0x06	0x20	F	32x16 bit signed fractional multiplication, higher signed 16 bits
MPYWHFMR	DOP	0x06	0x21	F	32x16 bit signed fractional multiplication with rounding, higher signed 16 bits
MACWHFM	DOP	0x06	0x22	F	32x16 bit signed fractional multiplication and accumulation, higher signed 16 bits
MACWHFMR	DOP	0x06	0x23	F	32x16 bit signed fractional multiplication and accumulation with rounding, higher signed 16 bits
MPYWHFL	DOP	0x06	0x24	F	32x16 bit signed fractional multiplication, lower signed 16 bits
MPYWHFLR	DOP	0x06	0x25	F	32x16 bit signed fractional multiplication with rounding, lower signed 16 bits
MACWHFL	DOP	0x06	0x26	F	Signed fractional multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. The result is saturated and rounded.
MACWHFLR	DOP	0x06	0x27	F	Signed 32x16 fractional multiplication and accumulation with rounding.
MACWHKL	DOP	0x06	0x28	F	Signed 32x16 integer multiplication and accumulation and shift-right the lower bits
MACWHKUL	DOP	0x06	0x29	F	Unsigned 32x16 integer multiplication and accumulation and shift-right the lower bits
MPYWHKL	DOP	0x06	0x2A	F	Signed integer multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator.

Table 7-52 32x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MPYWHKUL	DOP	0x06	0x2B	F	Unsigned integer multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator.
MSUBWHFL	DOP	0x06	0x14	F	Signed fractional multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. Subtract the product from the wide accumulator.
MSUBWHFLR	DOP	0x06	0x1A	F	Signed fractional multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. Subtract the product from the wide accumulator. Return the rounded and saturated result.
MSUBWHFM	DOP	0x06	0x2C	F	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand.
MSUBWHFMR	DOP	0x06	0x2D	F	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand; with rounded result

7.11.10 Complex Operations



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in “[Expansion of Literals in Vector Instructions](#)” on page 118. Short immediates are replicated for each vector lane.

Table 7-53 Complex Operations

Instruction	Type	Major	Minor	F bit	Description
CBFLYHF1R	SOP	0x06	0x19	0	16 + 16 bit complex FFT butterfly, second half;

Table 7-53 Complex Operations

Instruction	Type	Major	Minor	F bit	Description
CMPYHNFR	DOP	0x06	0x00	1	16 + 16 bit complex signed fractional multiplication with rounding, no fractional shift
CMPYHFR	DOP	0x06	0x01	1	16 + 16 bit complex signed fractional multiplication with rounding
CMPYHFR	DOP	0x06	0x02	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding
CMPYCHFR	DOP	0x06	0x05	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding
CMACHNFR	DOP	0x06	0x06	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding with no fractional shift
CMACHFR	DOP	0x06	0x07	1	16 + 16 bit complex signed fractional multiplication and accumulation with rounding
CMACCHNFR	DOP	0x06	0x08	1	16 + 16 bit complex conjugated signed fractional multiplication and accumulation with rounding with no fractional shift
CMACCHFR	DOP	0x06	0x09	1	16 + 16 bit complex conjugated signed fractional multiplication and accumulation with rounding
CMPYHFMR	DOP	0x06	0x1B	0	16 + 16 bit complex signed fractional multiplication with rounding
CBFLYHF0R	DOP	0x06	0x1B	1	16 + 16 bit complex FFT butterfly, first half

7.11.11 Vector 64-Bit Operations

Table 7-54 Vector 64-Bit Operations

Instruction	Type	Major	Minor	F bit	Description
DMPYHWF	DOP	0x6	0x3D		Sum of dual 16x16 signed fractional multiplication
QMACHF	DOP	0x6	0x3E		Quad 16x16 signed fractional multiply and accumulate; result is saturated
DMACWHF	DOP	0x6	0x3F		Dual 32x16 fractional multiply and accumulate
VADDS4H	DOP	0x5	0x38	1	Four-way 16-bit vector addition.
VSUBS4H	DOP	0x5	0x39	1	Four-way 16-bit vector subtraction
VADDSUBS4H	DOP	0x5	0x3A	1	Four-way 16-bit vector addition and subtraction.
VSUBADDS4H	DOP	0x5	0x3B	1	Four-way 16-bit vector subtraction and addition
VADDS2	DOP	0x5	0x3C	1	Two-way 32-bit vector addition.
VSUBS2	DOP	0x5	0x3D	1	Two-way 32-bit vector subtraction.
VADDSUBS	DOP	0x5	0x3E	1	Two-way 32-bit vector addition and subtraction

7.11.12 Divide and Square-Root Instructions



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in [“Expansion of Literals in Vector Instructions”](#) on page 118. Short immediates are replicated for each vector lane.

Table 7-55 Divide and Square-Root Operations

Instruction	Type	Major	Minor	F bit	Description
DIV	DOP	0x05	0x04	F	2's Complement Integer Divide
DIVU	DOP	0x05	0x05	F	Unsigned integer division
REM	DOP	0x05	0x08	F	2's complement integer Remainder

Table 7-55 Divide and Square-Root Operations

Instruction	Type	Major	Minor	F bit	Description
REMU	DOP	0x05	0x09	F	Unsigned integer Remainder
DIVF	DOP	0x06	0x10	F	Signed 32-bit fractional division
SQRT	SOP	0x05	0x30	F	Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result
SQRTF	SOP	0x05	0x31	F	Compute the square root of an unsigned Q31 fraction, yielding a Q31 fractional result.

7.11.13 Address Generation Instructions

Table 7-56 Address Generation Instructions

Instruction	Type	Major	Minor	F bit	Description
MODIF	ZOP	0x05	0x05	0	Apply modifier from immediate
MODAPP	XOP	0x05	0x03E	0	Apply modifier from register

8

32-bit Instruction Formats Reference

This chapter explains the 32-bit instruction formats of ARCV2 instruction set architecture. The chapter on “[Instruction Set Summary](#)” on page 299 lists and summarizes the 32-bit formats. In particular, [Table 7-4](#) on page 305 illustrates all of the 32-bit top-level formats of the ARCV2 ISA in a concise form, and may be used as a reference. The list of syntax conventions is shown in [Table 7-16](#) on page 314, and section “[Encoding Notation](#)” on page 306 describes the encoding notation used when describing instruction encodings in this chapter.

This chapter describes the following 32-bit ARCV2 instruction formats:

- [Branch Format \[F32_BR0\]](#)
- [BRcc, BBITn and BL Format \[F32_BR1\]](#)
- [Load Register with Offset Format \[F32_LD_OFFSET\]](#)
- [Store Register with Offset Format \[F32_ST_OFFSET\]](#)
- [General Operations Format \[F32_GEN4\]](#)
- [APEX Extension Instruction Format \[F32_APEX\]](#)

Each section provides details of the sub-formats, and describes how instructions and their operands are encoded in those sub-formats.

For a summary of the instructions discussed in this chapter, and an explanation of the functions they perform, see [Chapter 7, “Instruction Set Summary”](#).

For full details on the available encodings and syntax for each instruction, see to the relevant page for each instructions in [Chapter 10, “Instruction Set Details”](#).

8.1 Branch Format [F32_BR0]

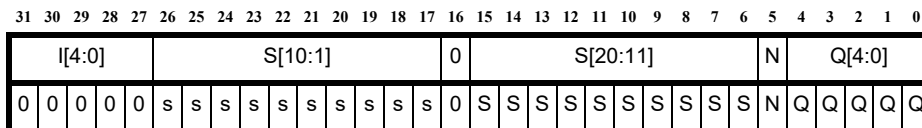
The F32_BR0 format is used to encode branch instructions with large displacements. There are two sub-formats, COND and UCOND_FAR. The COND format is used to encode conditional branch instructions, with an optional delay slot, and the UCOND_FAR format is used to encode unconditional branches with maximum possible displacement and an optional delay slot.

For all instructions in this format the branch target address is 16-bit aligned, as all ARCV2 instructions are 16-bit aligned in memory. Therefore bit 0 of the byte address offset is assumed to be zero, and is omitted from the instruction encoding.

8.1.1 Branch-Conditional Sub-format [F32_BR0, COND]

This format encodes conditional branch instructions with 21-bit relative branch offset. The encoding of condition code tests can be found in [Table 7-7](#) on page 308. For information about the encoding of delay-slot modes, see [Table 7-8](#) on page 309.

Figure 8-1 Branch Conditionally Format



Values 0x10 to 0x1F in the condition code field, Q, raise an [Illegal Instruction](#) exception, unless an extension condition code is defined.

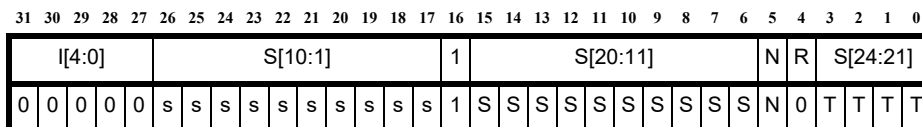
Syntax:

Bcc<.d> s21 (branch if condition is true)

8.1.2 Branch Unconditional Far Sub-format [F32_BR0, UCOND_FAR]

This format encodes an unconditional branch instruction with a 25-bit relative branch offset. See [Table 7-8](#) on page 309 for information on delay slot mode encoding.

Figure 8-2 Branch Unconditional Far Format



Syntax:

B<.d> s25 (unconditional branch far)

8.2 BRcc, BBITn and BL Format [F32_BR1]

This format encodes a set of branch instructions that include a comparison within the instruction. The BRcc instructions perform a relational test between two source operands and branch if the relation is true. The BBIT0 and BBIT1 instructions test one bit of the first operand, where the bit number is specified by the second operand, and then branch if the bit is 0 or 1 respectively.

These instructions may have an optional delay-slot. For information about the delay slot mode encoding, see [Table 7-8](#) on page 309.

These instructions also provide a way to encode a static branch prediction. This is a hint to hardware that, in the absence of a dynamic prediction, the branch is statistically more likely to go one way than the other. Not all implementations of the ARCV2 ISA exploit these hints, but compilers should attempt to provide the most appropriate hint to each BRcc or BBITn instruction, where possible. The encoding of static prediction hints is part of the instruction encoding for this group of instructions, and can be seen in “[Static Branch Prediction Mode](#)” on page 362

There are two operand formats for these instructions, one that compares two registers (REG_REG) and one that compares a register with an unsigned 6-bit literal (REG_U6).

8.2.1 BRcc / BBITn Register-register Format [F32_BR1, BCC, REG_REG]

For all instructions in this sub-format the branch target address is 16-bit aligned, as all ARCV2 instructions are 16-bit aligned in memory. Therefore bit 0 of the byte address offset is assumed to be zero, and is omitted from the instruction encoding.

Figure 8-3 Branch on Compare Register-register Format

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
I[4:0]				B[2:0]			S[7:1]							1	S8	B[5:3]			C[5:0]					N	0	i[3:0]					
0	0	0	0	1	b	b	b	s	s	s	s	s	s	s	1	S	B	B	B	C	C	C	C	C	C	N	0	i	i	i	i

Syntax:

BRcc<.d><.T> b,c,s9 (branch if reg-reg compare is true, swap regs if inverse condition required)

BRcc<.T> b,limm,s9 (branch if reg-limm compare is true)

BRcc<.T> limm,c,s9 (branch if limm-reg compare is true)

BBIT0<.d><.T> b,c,s9 (branch if bit c in reg b is clear)

BBIT0<.T> b,limm,s9 (branch if reg-limm compare is true)

BBIT0<.T> limm,c,s9 (branch if limm-reg compare is true)

BBIT1<.d><.T> b,c,s9 (branch if bit c in reg b is set)

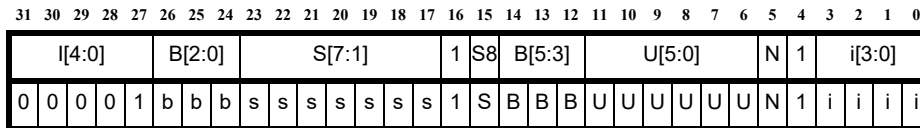
BBIT1<.T> b,limm,s9 (branch if reg-limm compare is true)

BBIT1<.T> limm,c,s9 (branch if limm-reg compare is true)

8.2.2 Register-immediate Format [F32_BR1, BCC, REG_U6]

For all instructions in this sub-format the branch target address is 16-bit aligned, as all ARCV2 instructions are 16-bit aligned in memory. Therefore bit 0 of the byte address offset is assumed to be zero, and is omitted from the instruction encoding.

Figure 8-4 Branch on Compare Register-Immediate Format



Syntax:

BRcc<.d><.T> b,u6,s9 (branch if reg-immediate compare is true, use "immediate+1" if a missing condition is required)

BBIT0<.d><.T> b,u6,s9 (branch if bit u6 in reg b is clear)

BBIT1<.d><.T> b,u6,s9 (branch if bit u6 in reg b is set)

8.2.2.1 Static Branch Prediction Mode

Table 7-9 on page 310 lists the interpretation of the <.T> syntax for statically-predicted branch instructions. For best performance, compilers or assembly programmers must ensure that the static branch prediction is set according to the most probable outcome of each branch, based either on profiling information or deductive analysis of the program. Each statically-predictable instruction has a prediction mode that is either BTFN or FTBN. These combine with the sign of the branch displacement to give a prediction of taken or not-taken. The BTFN mode predicts a branch to be “backwards taken, forwards not-taken”. Conversely, a FTBN mode predicts a branch to be “forwards taken, backwards not-taken”. The BTFN mode should normally be the default mode generated by a compiler in the absence of profiling information or static deductive reasoning about the branch direction. A BTFN prediction mode typically gives acceptable results on the majority of branches.

8.2.2.2 Encoding of BRcc, BBIT0, and BBIT1 Instructions

Table 8-1 Branch on Compare/Bit Test Register-Register

Sub-opcode i field (4 bits)	Instruction	Operation	Static Branch Prediction	Branch Condition Tested
0x00	BREQ	src1 – src2	BTFN	src1 is equal to src2
0x01	BRNE	src1 – src2	BTFN	src1 and src2 are not equal
0x02	BRLT	src1 – src2	BTFN	src1 is less than src2 (signed)
0x03	BRGE	src1 – src2	BTFN	src1 is greater or equal to src2 (signed)
0x04	BRLO	src1 – src2	BTFN	src1 is lower than src2 (unsigned)

Table 8-1 Branch on Compare/Bit Test Register-Register (Continued)

Sub-opcode i field (4 bits)	Instruction	Operation	Static Branch Prediction	Branch Condition Tested
0x05	BRHS	src1 – src2	BTFN	src1 higher than or same as src2 (unsigned)
0x06	BBIT0	(src1 and 2^{src2}) == 0	FTBN	bit src2 in src1 is clear
0x07	BBIT1	(src1 and 2^{src2}) == 0	FTBN	bit src2 in src1 is set
0x08	BREQ	src1 – src2	FTBN	src1 is equal to src2
0x09	BRNE	src1 – src2	FTBN	src1 and src2 are not equal
0x0A	BRLT	src1 – src2	FTBN	src1 is less than src2 (signed)
0x0B	BRGE	src1 – src2	FTBN	src1 is greater or equal to src2 (signed)
0x0C	BRLO	src1 – src2	FTBN	src1 is lower than src2 (unsigned)
0x0D	BRHS	src1 – src2	FTBN	src1 higher than or same as src2 (unsigned)
0x0E	BBIT0	(src1 and 2^{src2}) == 0	BTFN	bit src2 in src1 is clear
0x0F	BBIT1	(src1 and 2^{src2}) == 0	BTFN	bit src2 in src1 is set

8.2.3 Branch-and-link Format [F32_BR1, BL]

This sub-format encodes branch instructions that have return-address linkage to support function calls. As a side-effect of the branch, the BLINK register is assigned the address of the instruction following the branch, and its delay-slot if present, thus providing the function return address in the BLINK register.

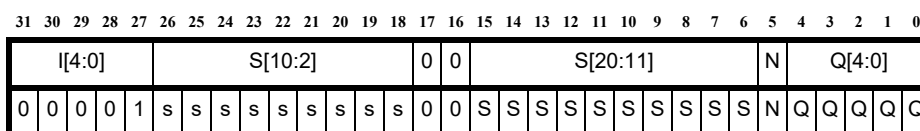
All function entry points are aligned to a 4-byte boundary, requiring that the target address is also 4-byte aligned. Thus the least-significant two bits of the branch offset can be assumed to be zero, and are omitted from the instruction encoding.

The branch instructions in this sub-format all encode a delay-slot mode, allowing an optional delay-slot instruction to be specified. For information about delay slot mode encoding, see [Table 7-8](#).

There are two sub-formats, a conditional format, COND, and an unconditional format that provides a greater branch displacement, UCOND_FAR. These are described in the following two subsections.

8.2.3.1 Branch-and-link Conditional Format [F32_BR1, BL, COND]

This sub-format encodes conditional branch-and-link instructions. For information about condition code test encoding, see [Table 7-7](#) on page 308.

Figure 8-5 Branch and Link Conditionally Format

Values 0x10 to 0x1F in the condition code field, Q, raise an **Illegal Instruction** exception, unless an extension condition code is defined.

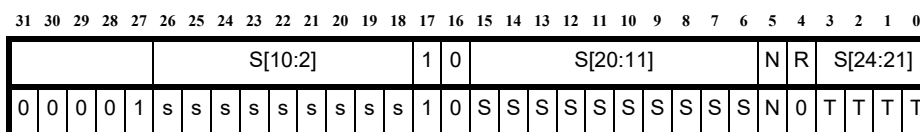
Syntax:

BLcc<.d> s21 (*branch if condition is true*)

8.2.3.2 Branch-and-link Unconditional Far Format [F32_BR1, BL, UCOND_FAR]

This format encodes unconditional branch-and-link instructions. The instruction bits not used to encode a condition field are used to provide a larger branch offset, giving a greater range for branch target addresses.

Figure 8-6 Branch and Link Unconditional Far Format



The reserved field, R, is ignored.

Syntax:

BL<.d> s25 (*unconditional branch far*)

8.3 Load Register with Offset Format [F32_LD_OFFSET]

For information about encoding the Load instruction, see [Table 7-10](#) on page 310, [Table 7-11](#) on page 311, [Table 7-13](#) on page 312, and [Table 7-14](#) on page 312.

Figure 8-7 Load Register with Offset Format

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
I[4:0]				B[2:0]			S[7:0]								S8	B[5:3]			Di	a	a	Z	Z	X	A[5:0]						
0	0	0	1	0	b	b	b	s	s	s	s	s	s	s	s	S	B	B	B	D	a	a	Z	Z	X	A	A	A	A	A	A

The program counter (PCL) is not permitted to be the destination of a load instruction. Values 0x3D and 0x3F in the destination register field, A, raise an **Illegal Instruction** exception.

The loop count register (LP_COUNT) is not permitted to be the destination of a load instruction, indicated by A=0x3C. Any use of LP_COUNT as the destination register raises an **Illegal Instruction** exception.

A value of 0x3 in the data size mode field, ZZ, raises an **Illegal Instruction** exception when LL64_OPTION is disabled.

The sign extension field, X, must not be set when the load is of 32-bit word data ZZ=0x0. This combination raises an **Illegal Instruction** exception.

Using incrementing addressing modes, with a long immediate value as the base register, is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, raises an **Illegal Instruction** exception.

Syntax:

LD<zz><.x><.aa><.di> a,[b,s9]

LD<zz><.x><.di> a,[limm,s9] (*use ld a,[limm]*)

LD<zz><.x><.di> a,[limm] (= *ld a,[limm,0]*)

8.4 Store Register with Offset Format [F32_ST_OFFSET]

For information about encoding the Store instruction, see [Table 7-10](#) on page 310, [Table 7-13](#) on page 312, and [Table 7-13](#) on page 312.

Figure 8-8 Store Register with Offset Format

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
I[4:0]				B[2:0]			S[7:0]								S8	B[5:3]			C[5:0]					Di	A	A	Z	Z	K		
0	0	0	1	1	b	b	b	s	s	s	s	s	s	s	s	S	B	B	B	C	C	C	C	C	C	D	a	a	Z	Z	0

Figure 8-9 Store Immediate with Offset Format

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	
I[4:0]				B[2:0]			S[7:0]								S8	B[5:3]			W[5:0]					Di	A	A	Z	Z	K			
0	0	0	1	1	b	b	b	s	s	s	s	s	s	s	s	S	B	B	B	w	w	w	w	w	w	w	D	a	a	Z	Z	1

A value of 0x3 in the data size mode field, ZZ, raises an [Illegal Instruction](#) exception when LL64_OPTION is disabled.

Using incrementing addressing modes, with a long immediate value as the base register, is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, raises an [Illegal Instruction](#) exception.

Syntax:

ST<zz><.aa><.di> c,[b,s9]

ST<zz><.di> c,[limm] (= st c,[limm,0])

ST<zz><.aa><.di> limm,[b,s9]

8.5 General Operations Format [F32_GEN4]

The ARCV2 instruction set defines a collection of hierarchical formats that are used to encode the majority of instructions that are neither branches nor load-store operations. These typically define a monadic or dyadic operator, which is applies to one or two source operands. Many, but not all, of these operators write a result to their destination, and they may also have the capability to set arithmetic status flags. Certain sub-formats allow a condition (also known as a predicate) to be specified, allowing the instruction to be executed only when the value of that condition is true.

8.5.1 Operator Format Indicators

The General Operations format sub-divides the encoding space of each top-level format into three distinct operator formats referred to as DOP, SOP and ZOP. Generally speaking, the DOP formats encode operators that have two source operands and one destination operand. Likewise, the SOP formats specify one source and one destination operand, and the ZOP formats specify zero or one operand, which may be a source or a destination (and occasionally both).

Table 8-2 illustrates the hierarchy within the General Operations formats. In these formats bit-positions [21:16], define the first operator field, and when this is not equal to 0x2F it specifies a DOP-format instruction. However, when the first operator is 0x2F it indicates a non-DOP instruction, and then the second operator field in bit-positions [5:0] comes into play. When the second operator is not equal to 0x3F it specifies a SOP-format instruction, and when the second operator is 0x3F it indicates a ZOP-format instruction. Bits [5:3] of each ZOP-format operator are located in bits [14:12] of the instruction, and bits [2:0] of the operator are located in bits [26:24] of the instruction.

Table 8-2 Hierarchical Sub-formats of the General Operations Formats

32-bit Format Hierarchy			Major Opcode	Layout of Instruction Formats																																				
(a)	(b)	(c)	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						
F32_GEN4	DOP	REG_REG	F32_GEN4				B[2:0]				00 01 10 11 i[5:0] except when (i[5:0] == 101111)				F				B[5:3]				C[5:0]				A[5:0]													
		REG_U6																					U[5:0]																	
		REG_S12																					S[5:0]												S[11:6]					
		COND_REG																					C[5:0]												0		Q[4:0]			
		COND_U6																					U[5:0]												1					
	SOP	REG_REG	F32_GEN4				B[2:0]				00 01 1 0 1 1 1 1				F				B[5:3]				C[5:0]				i[5:0] (except 111111)													
		REG_U6																					U[5:0]																	
	ZOP	REG_REG	F32_GEN4				i[2:0]				00 01 1 0 1 1 1 1				F				i[5:3]				C[5:0]				1 1 1 1 1 1													
		REG_U6																					U[5:0]																	

8.5.2 Operand Format Indicators

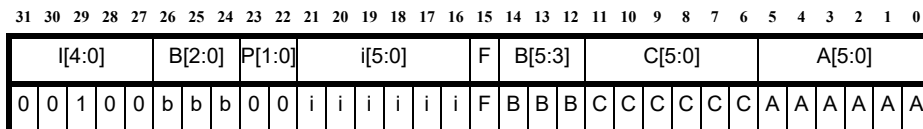
The General Operations formats provide five operand sub-formats, encoded with the P[1:0] and M fields, located in instruction bits [23:22] and [5] respectively, as shown in Table 8-3.

Table 8-3 Operand Sub-format Indicators

Format Name	P[1:0]	M	Operand Format Description
REG_REG	00	N/A	All operand fields specify registers
REG_U6IMM	01	N/A	Source 2 is a 6-bit unsigned immediate, other operands are registers
REG_S12IMM	10	N/A	Source 2 is a 12-bit signed immediate, other operands are registers
COND_REG	11	0	Conditional instruction. Destination (if any) is also source 1. Source 2 is a register
COND_REG_U6IMM	11	1	Conditional instruction. Destination (if any) is also source 1. Source 2 is a 6-bit unsigned immediate

8.5.3 Syntax and Encoding of Instructions in General Operations Formats

8.5.3.1 General Operations Register-register Format

Figure 8-10 General Operations Register-Register Format

Syntax:

dop<.f> a,b,c

dop<.f> a,limm,c (If B = 62)

dop<.f> a,b,limm (If C = 62)

dop<.f> a,limm,limm (If B = C = 62, not a useful format)

dop<.f> 0,b,c (If A = 62)

dop<.f> 0,limm,c (Redundant format, see [General Operations Conditional Register](#) format on [page 369](#))

dop<.f> 0,b,limm (If A = 62, B = 62)

dop<.f> 0,limm,limm (If A = 0, B = C = 62, not a useful format)

sop<.f> b,c

sop<.f> b,limm (If C = 62)

sop<.f> 0,c (If B = 62)

sop<.f> 0,limm (If B = C = 62, not a useful format)

zop<.f> c

zop<.f> limm (If C = 62)

8.5.3.2 General Operations Register with Unsigned 6-bit Immediate

Figure 8-11 General Operations Register with Unsigned 16-bit Immediate Format

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
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			U[5:0]					A[5:0]									
0	0	1	0	0	b	b	b	0	1	i	i	i	i	i	i	F	B	B	B	u	u	u	u	u	u	A	A	A	A	A	A

Syntax:

dop<.f> a,b,u6

dop<.f> a,limm,u6 (If B = 62, not useful format)

dop<.f> 0,b,u6 (If A = 62)

dop<.f> 0,limm,u6 (If A = B = 62, not useful format)

sop<.f> b,u6

sop<.f> 0,u6 (If B = 62)

zop<.f> u6

8.5.3.3 General Operations Register with Signed 12-bit Immediate

Figure 8-12 General Operations Register with Signed 12-bit Immediate Format

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
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			S[5:0]					S[11:6]									
0	0	1	0	0	b	b	b	1	0	i	i	i	i	i	i	F	B	B	B	s	s	s	s	s	s	S	S	S	S	S	S

A value of 0x2F in the first operator field, i, indicates a SOP or ZOP format instruction, which is invalid for this operand mode and raises an **Illegal Instruction** exception.

Syntax:

dop<.f> b,b,s12

dop<.f> 0,limm,s12 (If B = 62, not a useful format)

8.5.3.4 General Operations Conditional Register

Figure 8-13 General Operations Conditional Register Format

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
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			C[5:0]					M	Q[4:0]								

Figure 8-13 General Operations Conditional Register Format

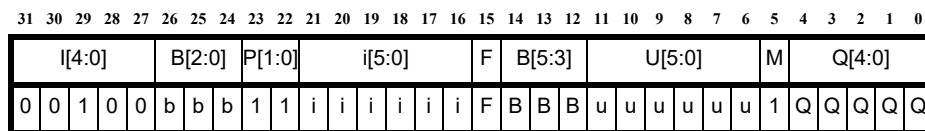
A value of 0x2F in the first operator field, i, indicates a SOP or ZOP format instruction, which is invalid for this operand mode and raises an [Illegal Instruction](#) exception.

Values 0x10 to 0x1F in the condition code field, Q, raise an [Illegal Instruction](#) exception, unless the Q field specifies a condition defined by an extension condition code.

Syntax:

```
dop<.cc><.f>      b,b,c
dop<.cc><.f>      0,limm,c      (If B = 62)
dop<.cc><.f>      b,b,limm      (If C = 62)
dop<.cc><.f>      0,limm,limm   (If B = C = 62, not a useful format)
```

8.5.3.5 General Operations Conditional Register with Unsigned 6-bit Immediate

Figure 8-14 General Operations Conditional Register with Unsigned 6-bit Immediate

A value of 0x2F in the first operator field, i, indicates a SOP or ZOP format instruction, which is invalid for this operand mode and raises an [Illegal Instruction](#) exception.

Values 0x10 to 0x1F in the condition code field, Q, raise an [Illegal Instruction](#) exception, unless the Q field specifies a condition defined by an extension condition code.

Syntax:

```
dop<.cc><.f>      b,b,u6
dop<.cc><.f>      0,limm,u6     (If B = 62, not a useful format)
```

8.5.4 Dual-operand Instructions, F32_GEN4

The DOP opcode assignments for F32_GEN4 are shown in Table 8-4. Subsequent sections contain similar tables for SOP and ZOP format instructions, and for other top-level formats (for example, F32_EXT5 and F32_EXT6). Each cell in these table represents a unique opcode slot that may be occupied by at most one instruction. Each table has eight columns, representing the eight value of the most significant three bits of the instruction opcode $i[5:3]$. Each table has eight pairs of rows, with each pair representing one of the eight values of the least significant three bits of the instruction opcode $i[5:3]$. The first row of each pair encodes instructions that have the $\langle.F\rangle$ bit set to 0, and the second row of the pair encodes instructions for which the $\langle.F\rangle$ bit is 1.

Any attempt to execute an instruction with an opcode marked **Illegal** in one of the opcode tables, raises an Illegal Instruction Exception.

A **Reserved** cell is one that is not allocated to an instruction, although attempting to execute the instruction with that opcode does not result in an exception. These cases indicate that the $\langle.F\rangle$ bit is ignored during the decoding of the instruction that is positioned directly above the **Reserved** cell.

Table 8-4 Opcode assignment for DOP instructions in major op-code 0x04 (F32_GEN4)

DOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ADD	MAX	BCLR	SUB2	J	LP	LD	SETEQ
	1	ADD.F	MAX.F	BCLR.F	SUB2.F	Reserved	Reserved		SETEQ.F
001	0	ADC	MIN	Illegal	SUB3	J.D	FLAG		SETNE
	1	ADC.F	MIN.F	BTST	SUB3.F	Reserved	KFLAG		SETNE.F
010	0	SUB	MOV	BXOR	MPYLO	JL	LR		SETLT
	1	SUB.F	MOV.F	BXOR.F	MPYLO.F	Reserved	Reserved		SETLT.F
011	0	SBC	Illegal	BMSK	MPYHI	JL.D	SR		SETGE
	1	SBC.F	TST	BMSK.F	MPYHI.F	Reserved	Reserved		SETGE.F
100	0	AND	DBNZ	ADD1	MPYHIU	BI	BMSKN		SETLO
	1	AND.F	CMP	ADD1.F	MPYHIU.F	Illegal	BMSKN.F		SETLO.F
101	0	OR	DBNZ.D	ADD2	MPYLOU	BIH	XBFU		SETHS
	1	OR.F	RCMP	ADD2.F	MPYLOU.F	Illegal	XBFU.F		SETHS.F
110	0	BIC	RSUB	ADD3	MPYW	LDI	Illegal		SETLE
	1	BIC.F	RSUB.F	ADD3.F	MPYW.F	Reserved	Illegal		SETLE.F
111	0	XOR	BSET	SUB1	MPYWU	AEX	SOP_FMT		SETGT
	1	XOR.F	BSET.F	SUB1.F	MPYWU.F	Reserved			SETGT.F

The F32_GEN4 format supports the majority of ALU operations in an orthogonal encoding of operand formats. It also contains certain jump and load instructions, on which are imposed a number of restrictions on the use of operand sub-formats. These **special formats** reserve certain operands or sub-formats that are not applicable to the instructions, or groups of instructions, encoded in those formats.

Table 8-5 Special DOP sub-formats in F32_GEN4

Special-format instructions		Major Opcode	Layout of Instruction Formats																																		
Instruction	i[5:0]	313029282726252423222120191817161514131211109876543210																																			
J<cc>	0x20	0x4				R				i[5:0]				R				R				C[5:0]				Reserved											
J<cc>.D	0x21																					00															
JL<cc>	0x22																					01				S[5:0]				S[11:6]							
JL<cc>.D	0x23																					10				C[5:0]				0				Q[4:0]			
		11				U[5:0]				1																											
LP<cc>	0x28	313029282726252423222120191817161514131211109876543210																																			
		0x4				R				i[5:0]				R				R				U[5:0]				Reserved											
																						00				S[5:0]				S[11:6]							
																						01				U[5:0]				1				Q[4:0]			
MOV<.cc><.F>	0x0A	313029282726252423222120191817161514131211109876543210																																			
BTST<.cc>	0x11 (F=1)	0x4				B[2:0]				i[5:0]				F				B[5:3]				C[5:0]				Reserved											
TST<.cc>	0x0B (F=1)																					00								S[5:0]				S[11:6]			
																						01				C[5:0]				0				Q[4:0]			
																						10				U[5:0]				1							
FLAG<.cc>	0x29 (F=0)	313029282726252423222120191817161514131211109876543210																																			
KFLAG<.cc>	0x29 (F=1)	0x4				R				i[5:0]				F				R				C[5:0]				Reserved											
																						00								U[5:0]							
																						01				S[5:0]				S[11:6]							
																						10				C[5:0]				0				Q[4:0]			
		11				U[5:0]				1																											
AEX<.cc>	0x27	313029282726252423222120191817161514131211109876543210																																			
CMP<.cc>	0x0C	0x4				B[2:0]				i[5:0]				R				B[5:3]				C[5:0]				Reserved											
RCMP<.cc>	0x0D																					00								U[5:0]							
																						01				S[5:0]				S[11:6]							
																						10				C[5:0]				0				Q[4:0]			
		11				U[5:0]				1																											
LR	0x2A	313029282726252423222120191817161514131211109876543210																																			
SR	0x2B	0x4				B[2:0]				i[5:0]				R				B[5:3]				C[5:0]				Reserved											
																						00								U[5:0]							
																						01				S[5:0]				S[11:6]							
LDI	0x26	0x4				B[2:0]				i[5:0]				R				B[5:3]				C[5:0]				Reserved											
																						00								U[5:0]							
		01				S[5:0]				S[11:6]																											
BI, BIH	0x24, 0x25	313029282726252423222120191817161514131211109876543210																																			
		0x4				R				00				i[5:0]				0				R				C[5:0]				Reserved							

8.5.5 Move and Compare Instructions, 0x04, [0x0A – 0x0D] and 0x04, [0x11]

The move and compare instructions (MOV, TST, CMP, RCMP, and BTST) use two operands. The destination field A is ignored for these instructions and instead the B and C fields are used accordingly.

8.5.6 Jump and Jump-and-Link Conditionally, 0x04, [0x20 – 0x23]

The jump (Jcc) and jump-and link (JLcc) instructions are specially encoded in major opcode 0x04 in which the B field is reserved and must be set to 0x0. Any value in the B field is ignored by the processor. The destination register, A field, must also be set to 0x0 when the operand mode, P, is 0x0 or 0x1. If P is 0x0 or 0x1, any value in the A field is ignored.

8.5.7 Load Register-Register, 0x04, [0x30 – 0x37]

Load register-register instruction, LD, is specially encoded in major opcode 0x04; the F and two mode bits usually found in the instruction word bit[15] and bits[23:22] are replaced by a D and two A bits. The normal “conditional/immediate” mode bits are replaced by addressing mode bits.

For information about encoding the Load instruction, see [Table 7-10](#) on page 310, [Table 7-14](#) on page 312, and [Table 7-13](#) on page 312.

Figure 8-15 Load Register-Register Format

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
I[4:0]				B[2:0]			A	A	1	1	0	Z	Z	X	Di	B[5:3]			C[5:0]					A[5:0]							
0	0	1	0	0	b	b	b	a	a	1	1	0	Z	Z	X	D	B	B	B	C	C	C	C	C	C	A	A	A	A	A	A

The program counter (PCL) is not permitted to be the destination of a load instruction. Values, 0x3D and 0x3F, in the destination register field, A, raise an [Illegal Instruction](#) exception.

The loop count register (LP_COUNT) is not permitted to be the destination of a load instruction, A=0x3C, and raises an [Illegal Instruction](#) exception.

A value of 0x3 in the data size mode field, ZZ, raises an [Illegal Instruction](#) exception when LL64_OPTION is disabled.

The sign extension field, X, must not be set when the load is of 32-bit word data ZZ=0x0. This combination raises an [Illegal Instruction](#) exception.

Using incrementing addressing modes in combination with a long immediate values as the base register is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, raises an [Illegal Instruction](#) exception.

Syntax:

LD<zz><.x><.aa><.di> a,[b,c]

LD<zz><.x><.aa><.di> a,[b,limm]

LD<zz><.x><.di> a,[limm,c]

8.5.8 Single Operand Instructions, F32_GEN4

The sub-opcode 2 (destination 'a' field) is reserved for defining single source operand instructions when sub-opcode 1 of 0x2F is used.

Table 8-6 Opcode assignment for SOP instructions in major op-code 0x04 (F32_GEN4)

SOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ASL	EXTW	LLOCK	Illegal	Illegal	Illegal	Illegal	Illegal
		ASL.F	EXTW.F	LLOCK.DI	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	ASR	ABS	SCOND	Illegal	Illegal	Illegal	Illegal	Illegal
	1	ASR.F	ABS.F	SCOND.DI	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	LSR	NOT	LLOCKD	Illegal	Illegal	Illegal	Illegal	Illegal
	1	LSR.F	NOT.F	LLOCKD.DI	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	ROR	RLC	SCONDD	Illegal	Illegal	Illegal	Illegal	Illegal
	1	ROR.F	RLC.F	SCONDD.DI	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	RRC	EX	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	RRC.F	EX.DI	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	SEXB	ROL	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	SEXB.F	ROL.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	SEXW	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	SEXW.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	EXTB	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	ZOP_FMT
	1	EXTB.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	

8.5.9 Zero Operand Instructions, F32_GEN4

The sub-opcode 3 (source operand b field) is reserved for defining zero operand instructions when sub-opcode 2 of 0x3F is used.

Table 8-7 Opcode assignment for ZOP instructions in major op-code 0x04 (F32_GEN4)

ZOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	Illegal	WEVT	Illegal	NEXTTASK	Illegal	Illegal	Illegal	Illegal
		Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	SLEEP	WLFC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	SWI	DSYNC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

Table 8-7 Opcode assignment for ZOP instructions in major op-code 0x04 (F32_GEN4)

011	0	SYNC	DMB	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	RTIE	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	BRK	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	SETI	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	CLRI	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

Syntax:

SLEEP

SLEEP u6

SLEEP c

SWI

SYNC

RTIE

BRK *(Encoded as REG_U6IMM, but the redundant REG_REG format is also valid. See [Table 8-3](#) on page 368)*

SETI c

SETI u6

CLRI c

CLRI u6

8.6 F32_EXT5

8.6.1 Dual-operand Instructions, F32_EXT5

Table 8-8 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5)

opcode		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111

Table 8-8 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5)

000	0	ASL multiple	REM	DMPYH	MPYD	VMPY2HWF	DMPYHWF	Illegal	Illegal
	1	ASL multiple.F	REM.F	DMPY.F	MPYD.F	SJLI	DMPYHWF.F	Illegal	Illegal
001	0	LSR multiple	REMU	DMPYHU	MPYDU	VASL2H	VPACK2HL	Illegal	Illegal
	1	LSR multiple.F	REMU.F	DMPYHU.F	MPYDU.F	VASLS2H	VPACK2HM	Illegal	Illegal
010	0	ASR multiple	ASLS	DMACH	MACD	VASR2H	DMPYHF	Illegal	Illegal
	1	ASR multiple.F	ASLS.F	DMACH.F	MACD.F	VASRS2H	DMPYHF.F	Illegal	Illegal
011	0	ROR multiple	ASRS	DMACHU	MACDU	VLSR2H	DMPYHFR	Illegal	Illegal
	1	ROR multiple.F	ASRS.F	DMACHU.F	MACDU.F	VASRSR2H	DMPYHFR.F	Illegal	Illegal
100	0	DIV	ASRSR	VADD2H	VMPY2H	VADD4B	DMACHF	Illegal	Illegal
	1	DIV.F	ASRSR.F	VADDS2H	VMPY2HF	VMAX2H	DMACHF.F	Illegal	Illegal
101	0	DIVU	VALIGN2H	VSUB2H	VMPY2HU	VSUB4B	DMACHFR	Illegal	Illegal
	1	DIVU.F	SETACC	VSUBS2H	VMPY2HFR	VMIN2H	DMACHFR.F	Illegal	Illegal
110	0	ADDS	MAC	VADDSUB2H	VMACH2H	ADCS	VPERM	Illegal	Illegal
	1	ADDS.F	MAC.F	VADDSUBS2H	VMAC2HF	ADCS.F	BSPUSH	Illegal	Illegal
111	0	SUBS	MACU	VSUBADD2H	VMACH2HU	SBCS	SOP_FMT	Illegal	Illegal
	1	SUBS.F	MACU.F	VSUBADDS2H	VMAC2HFR	SBCS.F		Illegal	Illegal

8.6.2 Single-operand Instructions, F32_EXT5

Table 8-9 Opcode assignment for SOP instructions in major op-code 0x05 (F32_EXT5)

SOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	SWAP	NORMH	ROL8	GETACC	VEXT2BHLLF	VABS2H	SQRT	Illegal
		SWAP.F	NORMH.F	ROL8.F	Illegal	Illegal	Illegal	SQRT.F	Illegal
001	0	NORM	SWAPE	ROR8	NORMACC	VEXT2BHMLF	VABSS2H	SQRTF	Illegal
	1	NORM.F	SWAPE.F	ROR8.F	Illegal	Illegal	Illegal	SQRTF.F	Illegal
010	0	SATH	LSL16	FFS	SATF	VREP2H	VNEG2H	Illegal	Illegal
	1	SATH.F	LSL16.F	FFS.F	SATF.F	Illegal	Illegal	Illegal	Illegal
011	0	RNDH	LSR16	FLS	Illegal	VREP2HM	VNEGS2H	Illegal	Illegal
	1	RNDH.F	LSR16.F	FLS.F	Illegal	Illegal	Illegal	Illegal	Illegal

Table 8-9 Opcode assignment for SOP instructions in major op-code 0x05 (F32_EXT5)

100	0	ABSSH	ASR16	Illegal	VPACK2HBL	VEXT2BHL	VNORM2H	Illegal	Illegal
	1	ABSSH.F	ASR16.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	ABSS	ASR8	Illegal	VPACK2HBM	VEXT2BHM	Illegal	Illegal	Illegal
	1	ABSS.F	ASR8.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	NEGHS	LSR8	Illegal	VPACK2HBLF	VSEXT2BHL	BSPEEK	Illegal	Illegal
	1	NEGHS.F	LSR8.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	NEGS	LSL8	Illegal	VPACK2HBMF	VSEXT2BHM	PSPOP	Illegal	ZOP_FMT
	1	NEGS.F	LSL8.F	Illegal	Illegal	Illegal	Illegal	Illegal	

8.6.3 Zero-operand Instructions, F32_EXT5

Table 8-10 Opcode assignment for ZOP instructions in major op-code 0x05 (F32_EXT5)

ZOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ASLACC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	ASLSACC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	FLAGACC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	MODIF	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

8.7 F32_EXT6

8.7.1 Dual-operand Instructions, F32_EXT6

Table 8-11 Opcode assignment for DOP instructions in major op-code 0x06 (F32_EXT6)

opcode		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	FSMUL	FCVT32	DIVF	DMACHBL	MPYWHFM	MACWHKL	FDMUL	FCVT64
	1	CMPLYHNFR	CMACCHNFR	DIVF.F	DMACHBL.F	MPYWHFM.F	MACWHKL.F	Illegal	Illegal
001	0	FSADD	FCVT32_64	VMAC2HNFR	DMACHBM	MPYWHFMR	MACWHKUL	FDADD	FCVT64_32
	1	CMPLYHFR	CMACCHFR	VMSUB2HNFR	DMACHBM.F	MPYWHFMR.F	MACWHKUL.F	Illegal	Illegal
010	0	FSSUB	MPYF	MPYDF	MSUBWHFLR	MACWHFM	MPYWHKL	FDSUB	Illegal
	1	CMPYCHNFR	MPYF.F	MPYDF.F	MSUBWHFLR.F	MACWHFM.F	MPYWHKL.F	Illegal	Illegal
011	0	VMSUB2HFR	MPYFR	MACDF	CMPLYHFR	MACWHFMR	MPYWHKUL	Illegal	Illegal
	1	FSCMP	MPYFR.F	MACDF.F	CBFLYHFOR	MACWHFMR.F	MPYWHKUL.F	FDCMP	Illegal
100	0	VMSUB2HF	MACF	MSUBWHFL	MPYWHL	MPYWHFL	MSUBWHFM	Illegal	Illegal
	1	FSCMPF	MACF.F	MSUBWHFL.F	MPYWHL.F	MPYWHFL.F	MSUBWHFM.F	Illegal	Illegal
101	0	FSMADD	MACFR	MSUBDF	MACWHL	MPYWHFLR	MSUBWHFMR	Illegal	Illegal
	1	CMPYCHFR	MACFR.F	MSUBDF.F	MACWHL.F	MPYWHFLR.F	MSUBWHFMR.F	FDCMPF	Illegal
110	0	FSMSUB	MSUBF	DMPYHBL	MPYWHUL	MACWHFL	Illegal	FDMADD	Illegal
	1	CMACHNFR	MSUBF.F	DMPYHBL.F	MPYWHUL.F	MACWHFL.F	Illegal	Illegal	Illegal
111	0	FSDIV	MSUBFR	DMPYHBM	MACWHUL	MACWHFLR	SOP_FMT	FDDIV	Illegal
	1	CMACHFR	MUSBFR.F	DMPYHBM.F	MACWHUL.F	MACWHFLR.F		Illegal	Illegal

8.7.2 Single-operand Instructions, F32_EXT6

Table 8-12 Opcode assignment for SOP instructions in major op-code 0x06 (F32_EXT6)

SOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	FSSQRT	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	FDSQRT	Illegal	Illegal	CBFLYHF1R	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

Table 8-12 Opcode assignment for SOP instructions in major op-code 0x06 (F32_EXT6)

100	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	ZOP_FMT
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	

8.7.3 Zero-operand Instructions, F32_EXT6

Table 8-13 Opcode assignment for ZOP instructions in major op-code 0x06 (F32_EXT6)

ZOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	SFLAG	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
		Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

8.8 APEX Extension Instruction Format [F32_APEX]

The F32_APEX top-level format, assigned to major opcode 0x07, is available for use by User Extension Instructions defined using the APEX extension mechanism.

Figure 8-16 using major opcode 0x07 as an example to show the syntax of op<.f> a,b,c encoding.

Figure 8-16 Extension ALU Operation, register-register

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
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			C[5:0]					A[5:0]									
0	0	1	1	1	b	b	b	0	0	i	i	i	i	i	i	F	B	B	B	C	C	C	C	C	C	A	A	A	A	A	A

8.8.1 Extension ALU Operation, Register with Unsigned 6-bit Immediate

Figure 8-17 shows the APEX encoding syntax of op<.f> a,b,u6 encoding.

Figure 8-17 Extension ALU Operation, register with unsigned 6-bit immediate

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
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			U[5:0]					A[5:0]									
0	0	1	1	1	b	b	b	0	1	i	i	i	i	i	i	F	B	B	B	u	u	u	u	u	u	A	A	A	A	A	A

8.8.2 Extension ALU Operation, Register with Signed 12-bit Immediate

Using major opcode 0x05 as an example, Figure 8-18 shows the syntax of op<.f> b,b,s12 encoding.

Figure 8-18 Extension ALU Operation, register with signed 12-bit immediate

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
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			S[5:0]					S[11:6]									
0	0	1	1	1	b	b	b	1	0	i	i	i	i	i	i	F	B	B	B	s	s	s	s	s	s	S	S	S	S	S	S

A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and raises an [Illegal Instruction](#) exception.

8.8.3 Extension ALU Operation, Conditional Register

Figure 8-19 shows the syntax of op<.cc><.f> b,b,c that is encoded.

Figure 8-19 Extension ALU Operation, conditional register

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
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			C[5:0]					M	Q[4:0]								
0	0	1	1	1	b	b	b	1	1	i	i	i	i	i	i	F	B	B	B	C	C	C	C	C	C	0	Q	Q	Q	Q	Q

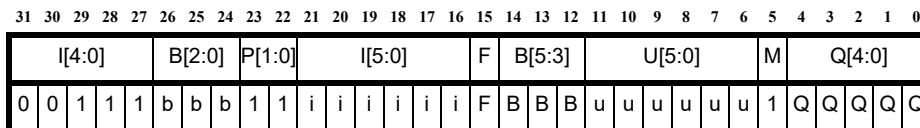
A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and raises an [Illegal Instruction](#) exception.

Values 0x10 to 0x1F in the condition code field, Q, raises an **Illegal Instruction** exception unless the Q field specifies a condition code defined as an extension condition.

8.8.4 Extension ALU Operation, Conditional Reg with Unsigned 6-bit Immediate

Using major opcode 0x05 as an example, [Figure 8-20](#) shows the syntax of op<.cc><.f> b,b,u6 encoding.

Figure 8-20 Extension ALU Operation, cc register with unsigned 6-bit immediate



A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and raises an **Illegal Instruction** exception.

Values 0x10 to 0x1F in the condition code field, Q, raise an **Illegal Instruction** exception unless the Q field specifies a condition code defined as an extension condition.

8.8.5 FastMath Extension Pack Instructions

[Table 8-14](#) lists the FastMath Pack Extension Instructions. If this extension is available with your processor, the instructions are available in the *Part: FastMath Pack Extension* in the *ARCV2 ISA Programmer's Reference Manual*.

Table 8-14 List of FastMath extension pack instructions

Instruction	Operation
FMP_ADDS	32-bit signed addition with result saturation
FMP_RNDH	Round and saturate a 32-bit signed 2's complement value to a 16-bit value
FMP_SATH	Saturate a 32-bit signed 2's complement integer to a 16-bit value
FMP_DIVF	Fractional division of Q31 numerator X, by Q31 divisor Y, where $\text{abs}(X) < \text{abs}(Y)$ or $X = -Y$, and $Y \neq 0$.
FMP_DIVF15	Fractional division of Q15 numerator X, by Q15 divisor Y, where $\text{abs}(X) < \text{abs}(Y)$ or $X = -Y$, and $Y \neq 0$.
FMP_RECIP	Computes the reciprocal of a Q31 fraction X, provided $X \neq 1$.
FMP_RECIP15	Computes the reciprocal of a Q15 fraction X, provided $X \neq 1$.
FMP_SQRTF	Computes the square root of the Q31 fractional argument X, when $X \geq 0$.
FMP_SQRTF15	Computes the square root of the Q15 fractional argument X, when $X \geq 0$.
FMP_COS	Computes the cosine function on the Q31 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting cosine value is returned as the Q31 result.

Table 8-14 List of FastMath extension pack instructions

Instruction	Operation
FMP_COS15	Computes the cosine function on the Q15 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting cosine value is returned as the Q15 result.
FMP_SIN	Computes the sine function on the Q31 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting sine value is returned as the Q31 result.
FMP_SIN15	Computes the sine function on the Q15 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting sine value is returned as the Q15 result.
FMP_ATAN	Computes the inverse tangent function on the Q31 fractional input operand. The input operand represents a real-valued tangent in the range $[-1,0)$ and the Q31 result represents the angle expressed in radians divided by Pi, for which the tangent is equal to the input operand.
FMP_ATAN15	Computes the inverse tangent function on the Q15 fractional input operands. The input operand represents a real-valued tangent in the range $[-1,0)$ and the Q15 result represents the angle expressed in radians divided by Pi, for which the tangent is equal to the input operand.
FMP_LOG2	Computes the base-2 logarithm of the Q31 fractional input operand X, when X is in the range $[0.5,1)$. The result $Y = \log_2(X)$ is a Q31 fraction in the range $[-1,0)$.
FMP_LOG215	Computes the base-2 logarithm of the Q15 fractional input operand X, when X is in the range $[0.5,1)$. The result $Y = \log_2(X)$ is a Q15 fraction in the range $[-1,0)$.
FMP_EXP2	Computes the base-2 exponentiation of the Q31 fractional input operand X, when X is in the range $[-1,0)$. The result $Y = 2^X$ is a Q31 fraction in the range $[0.5,1)$.
FMP_EXP215	Computes the base-2 exponentiation of the Q15 fractional input operand X, when X is in the range $[-1,0)$. The result $Y = 2^X$ is a Q15 fraction in the range $[0.5,1)$.

8.8.6 CryptoPack Instructions

The processor supports CryptoPack Extension Instructions. If this extension is available with your processor, the instructions are available in the *DesignWare ARC EM CryptoPack Reference* available with the CryptoPack media package.

9

16-bit Instruction Formats Reference

This chapter explains the 16-bit encoding formats for ARCV2 instructions. Most of the 16-bit formats redundantly encode restricted versions of instructions that are also defined in 32-bit formats. This enables more compact instructions to be selected in many cases. [Table 7-4](#) on page 305 illustrates all of the 16-bit top-level formats of the ARCV2 ISA in a concise form, and may be used as a reference. The list of syntax conventions is shown in [Table 7-16](#) on page 314, and section “[Encoding Notation](#)” on page 306 describes the encoding notation used when describing instruction encodings in this chapter.

This chapter describes each of the following 16-bit top-level formats:

- [Compact Move/Load \[F16_COMPACT\]](#)
- [Compact Load/Add/Sub \[F16_LD_ADD_SUB\]](#)
- [Compact Load/Store \[F16_LD_ST_1\]](#)
- [Indexed Jump or Execute \[F16_JLI_EI\]](#)
- [Load / Add Register-Register \[F16_LD_ADD_RR\]](#)
- [Add/Sub/Shift Register-Immediate \[F16_ADD_IMM\]](#)
- [Dual Register Operations \[F16_OP_HREG\]](#)
- [General Register Format Instructions \[F16_GEN_OP\]](#)
- [16-bit Load and Store Formats with Offset](#)
- [Shift/Subtract/Bit Immediate \[F16_SH_SUB_BIT\]](#)
- [Stack-based Operations \[F16_SP_OPS\]](#)
- [Load/Add GP-Relative \[F16_GP_LD_ADD\]](#)
- [Load PCL-Relative \[F16_PCL_LD\]](#)
- [Move Immediate \[F16_MV_IMM\]](#)
- [ADD/CMP Immediate \[F16_OP_IMM\]](#)
- [Branch on Compare Register with Zero \[F16_BCC_REG\]](#)
- [Branch Conditionally \[F16_BCC\]](#)
- [Branch and Link Unconditionally \[F16_BL\]](#)

Each section provides details of the sub-formats, and describes how instructions and their operands are encoded in those sub-formats.

For a summary of the instructions discussed in this chapter, and an explanation of the functions they perform, see [Chapter 7, “Instruction Set Summary”](#).

For full details on the available encodings and syntax for each instruction, see the relevant page for each instructions in [Chapter 10, “Instruction Set Details”](#).

9.1 Compact Move/Load [F16_COMPACT]

Extension Group: BASELINE

Figure 9-1 Compact Move Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					g[2:0]			h[2:0]			G[4:3]		i	H[4:3]	
0	1	0	0	0	g	g	g	h	h	h	G	G	0	H	H

This compact move instruction encodes the specifics required to move the content of one h-register to another. These h-registers specify one of registers R0-R31, with the exception of R29 and R30. R30 is actually a long-immediate data operand when used as the source register, and a null destination register.

Syntax:

MOV_S g,h

Extension Group: CODE_DENSITY

Figure 9-2 Compact Load Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					U	r[1:0]		h[2:0]			U[3:2]		i	H[4:3]	
0	1	0	0	0	U[4]	r	r	h	h	h	u	u	1	H	H

The compact load encoded in this format allows a load from h-register base address, plus a u5 scaled offset, into one of R0, R1, R2 or R3.

Syntax:

LD_S R0/1/2/3, [h,u5]

9.2 Compact Load/Add/Sub [F16_LD_ADD_SUB]

Extension Group: CODE_DENSITY

Figure 9-3 Compact Load and Subtract Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
l[4:0]					b[2:0]			c[2:0]			i	0	a[2:0]		
0	1	0	0	1	b	b	b	c	c	c	i	0	a	a	a

The first sub-format of opcode 0x09 encodes additional LD_S.AS and SUB_S instructions in a compact format to improve code density.

Syntax:

LD_S.AS a, [b, c]

SUB_S a, b, c

Figure 9-4 Compact Add Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
l[4:0]					b[2:0]			r	U[5:3]			1	u[2:0]		
0	1	0	0	1	b	b	b	r	U	U	U	1	u	u	u

The second sub-format of opcode 0x09 encodes additional ADD_S instructions, with R0 or R1 as their destination register, in a compact form to improve code density.

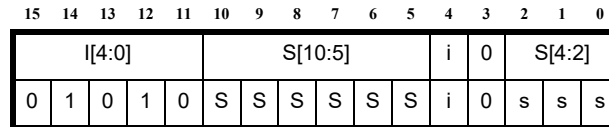
Syntax:

ADD_S r, b, u6 r can be R0 or R1

9.3 Compact Load/Store [F16_LD_ST_1]

Extension Group: CODE_DENSITY

Figure 9-5 Compact Load and Store Format



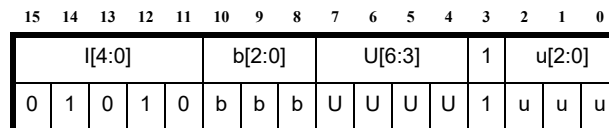
The first sub-format in opcode 0x0A encodes two additional GP-relative load/store instructions for use when accessing the most frequently occurring small data objects.

Syntax:

LD_S R1, [GP, s11]

ST_S R0, [GP, s11]

Figure 9-6 Compact Load Indexed Format



The second sub-format in opcode 0x0A encodes the Load Indexed instruction (LDI_S). This instruction loads from a table of 128 32-bit word entries, given the table index in the instruction. The intent of this instruction is to create a global constant pool to store the most commonly-used L IMM constants of the program. In this way, such constants only need to appear once in the linked executable, and such a constant can always be loaded into a compressed register using a 16-bit instruction.

LDI_BASE is an auxiliary register containing the address of the data table, which must be aligned on a 4-byte boundary.

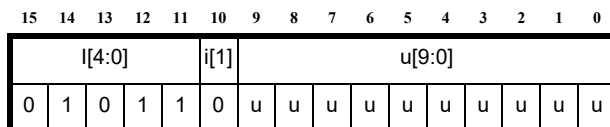
Syntax:

LDI_S b, [u7]

9.4 Indexed Jump or Execute [F16_JLI_EI]

Extension Group: CODE_DENSITY

Figure 9-7 Indexed Jump and Execute Format



Format 0x0B encodes indexed jump and branch operations. The JLI_S instruction is encoded in this format, and enables compact 16-bit encodings for function calls regardless of their relative distance from the current PC. A table of unconditional branch instructions, with an entry for each frequently referenced function, is placed at a location in memory pointed to by the JLI_BASE auxiliary register. The JLI_S instruction jumps to the location in the table selected by the u10 operand, and saves the next PC to the BLINK register, as do all Jump-and-link instructions.

Syntax:

JLI_S u10

Format 0x0B also encodes the Execute Indexed (EI_S) instruction. This encoding enables common instances of the same 32-bit encoded instruction to share a single copy of that instruction, located in a common instruction table pointed to by the EI_BASE auxiliary register.

Syntax:

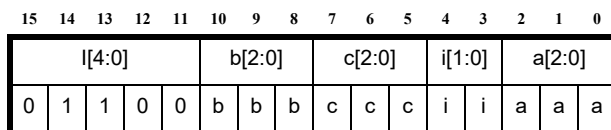
EI_S u10

Table 9-1 16-Bit Indexed Jump and Link Operations

Sub-opcode i field (1 bit)	Instruction	Operation	Description
0x0	JLI_S	BLINK ← next PC PC ← JLI_BASE + (u10 << 2)	Jump and link, to indexed table location
0x1	EI_S	STATUS32.ES ← 1 BTA ← next PC PC ← EI_BASE + (u10 << 2)	Execute instruction at table index

9.5 Load /Add Register-Register [F16_LD_ADD_RR]

Figure 9-8 Load and Add Register-Register Format



Syntax:

LD_S a, [b, c]

LDB_S a, [b, c]

LDH_S a, [b, c]

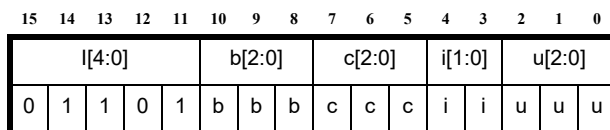
ADD_S a, b, c

Table 9-2 16-Bit, LD / ADD Register-Register

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	LD_S	$a \leftarrow \text{mem}[b + c].l$	Load 32-bit word (reg.+reg.)
0x01	LDB_S	$a \leftarrow \text{mem}[b + c].b$	Load unsigned byte (reg.+reg.)
0x02	LDH_S	$a \leftarrow \text{mem}[b + c].w$	Load unsigned 16-bit half-word (reg.+reg.)
0x03	ADD_S	$a \leftarrow b + c$	Add

9.6 Add/Sub/Shift Register-Immediate [F16_ADD_IMM]

Figure 9-9 Add/Sub/Shift Register-Immediate Format



Syntax:

ADD_S c, b, u3

SUB_S c, b, u3

ASL_S c, b, u3

ASR_S c, b, u3

Table 9-3 16-Bit, ADD/SUB Register-Immediate

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	ADD_S	$c \leftarrow b + u3$	Add
0x01	SUB_S	$c \leftarrow b - u3$	Subtract
0x02	ASL_S	$c \leftarrow b \text{ asl } u3$	Multiple arithmetic shift left
0x03	ASR_S	$c \leftarrow b \text{ asr } u3$	Multiple arithmetic shift right

In this format, the u3 operand encoding 000 is reserved, and must not be used by programmers or compilers.

All other values of the u3 operand are encoded as unsigned 3-bit binary values from 1 to 7 inclusive.

9.7 Dual Register Operations [F16_OP_HREG]

Figure 9-10 Dual Register with a 3-bit b Register Format

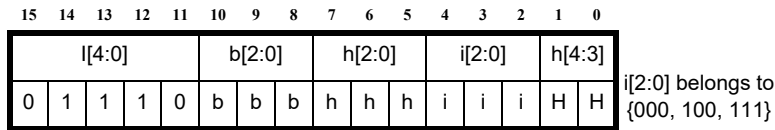
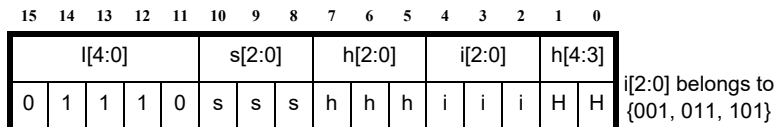


Figure 9-11 Dual Register with a 3-bit Biased Signed Integer Format



Format 0x0E encodes eight distinct operators, each with two register operands. One of the two operands is a 3-bit 'b' register, or a 3-bit biased signed integer 's3', and the other operand is a 5-bit 'h' register specifier. The 'b' register selects one of the registers normally accessible with a 3-bit register field within a 16-bit instruction encoding. The 'h' register is capable of selecting any register in the range r0 – r28, and also r31. The 'h' field encoding for r30 indicates the use of long immediate data. An 'h' field encoding of r29 is not allowed.

The s3 field encodes the signed integers values -1, 0, 1, 2, 3, 4, 5, 6 using binary codes 111, 000, 001, 010, 011, 100, 101, and 110, respectively.

Syntax:

ADD_S	b, b, h	(h != 0x1e)
ADD_S	b, b, limm	(h = 0x1e)
ADD_S	h, h, s3	(h != 0x1e)
ADD_S	0, limm, s3	(h = 0x1e)
MOV_S	h, s3	(h != 0x1e)
MOV_S	0, s3	(h = 0x1e)
MOV_S.NE	b, h	(h != 0x1e)
MOV_S.NE	b, limm	(h = 0x1e)
CMP_S	b, h	(h != 0x1e)
CMP_S	b, limm	(h = 0x1e)
CMP_S	h, s3	(h != 0x1e)
CMP_S	limm, s3	(h = 0x1e)

Table 9-4 16-Bit MOV/CMP/ADD with High Register

Sub-opcode i field (3 bits)	Instruction	Operation	Description
0x00	ADD_S	$b \leftarrow b + h$	Add
0x01	ADD_S	$h \leftarrow h + s3$	Add
0x02		Illegal Instruction Error	Reserved
0x03	MOV_S	$h \leftarrow s3$	Move
0x04	CMP_S	$b - h$	Compare
0x05	CMP_S	$h - s3$	Compare
0x06		Illegal Instruction Error	Reserved
0x07	MOV_S.NE	if (!Z) $b \leftarrow h$	Move if STATUS32[Z] == 0

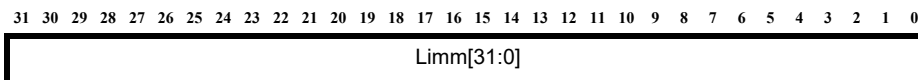
9.7.1 Long Immediate Operands in Format 0x0E

The special encoding of 0x1E for the 5-bit H register field in this instruction indicates a long immediate operand rather than a normal register. This encoding is similar to the usual 6-bit encoding of a limm data operand except for the absence of the most-significant 1.

The encoding 0x1E for LIMM data also means that register r30 is not available in this format. When a source register is set to r30, an explicit long immediate value follows the instruction word.

When a destination register is set to r30, there is no destination for the result of the instruction. Therefore, the result is discarded.

When an instruction uses long immediate, the first 16-bit half-word contains the instruction itself, and the subsequent 32-bit word is the long immediate (limm) data itself.

Figure 9-12 Long Immediate Operand Format

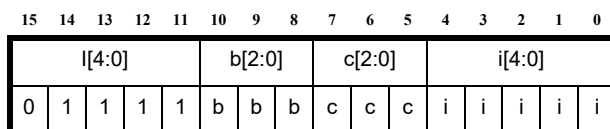
Syntax:

limm

9.8 General Register Format Instructions [F16_GEN_OP]

9.8.1 DOP-format General Operations

Figure 9-13 Compact General Operations Register-Register Format



Syntax:

op_S b,b,c

op_S b,c

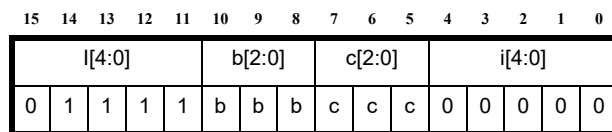
Table 9-5 DOP_format 16-Bit General Operations

Sub-opcode i field (5 bits)	Instruction	Operation	Description
0x00	SOPs	c field is sub-opcode2	See Table 9-6 on page 395.
0x01		Illegal Instruction	Reserved
0x02	SUB_S	$b \leftarrow b - c$	Subtract
0x03		Illegal Instruction	Reserved
0x04	AND_S	$b \leftarrow b \text{ and } c$	Logical bitwise AND
0x05	OR_S	$b \leftarrow b \text{ or } c$	Logical bitwise OR
0x06	BIC_S	$b \leftarrow b \text{ and not } c$	Logical bitwise AND with invert
0x07	XOR_S	$b \leftarrow b \text{ exclusive-or } c$	Logical bitwise exclusive-OR
0x08		Illegal Instruction	Reserved
0x09	MPYW_S	$b \leftarrow b * c$	Multiply 16-bit half-words
0x0A	MPYUW_S	$b \leftarrow b * c$ (unsigned)	Multiply unsigned 16-bit half-words
0x0B	TST_S	$b \text{ and } c$	Test
0x0C	MPY_S	$b \leftarrow b * c$	32 X 32 Multiply (least-significant half)
0x0D	SEXB_S	$b \leftarrow \text{sexb } c$	Sign extend byte
0x0E	SEXH_S	$b \leftarrow \text{sexw } c$	Sign extend 16-bit half-word
0x0F	EXTB_S	$b \leftarrow \text{extb } c$	Zero extend byte

Table 9-5 DOP_format 16-Bit General Operations (Continued)

Sub- opcode i field (5 bits)	Instruction	Operation	Description
0x10	EXTH_S	$b \leftarrow \text{extw } c$	Zero extend 16-bit half-word
0x11	ABS_S	$b \leftarrow \text{abs } c$	Absolute
0x12	NOT_S	$b \leftarrow \text{not } c$	Logical NOT
0x13	NEG_S	$b \leftarrow \text{neg } c$	Negate
0x14	ADD1_S	$b \leftarrow b + (c \ll 1)$	Add with left shift by 1
0x15	ADD2_S	$b \leftarrow b + (c \ll 2)$	Add with left shift by 2
0x16	ADD3_S	$b \leftarrow b + (c \ll 3)$	Add with left shift by 3
0x17		Illegal Instruction	Reserved
0x18	ASL_S	$b \leftarrow b \text{ asl } c$	Multiple arithmetic shift left
0x19	LSR_S	$b \leftarrow b \text{ lsr } c$	Multiple logical shift right
0x1A	ASR_S	$b \leftarrow b \text{ asr } c$	Multiple arithmetic shift right
0x1B	ASL_S	$b \leftarrow c + c$	Arithmetic shift left by one
0x1C	ASR_S	$b \leftarrow c \text{ asr } 1$	Arithmetic shift right by one
0x1D	LSR_S	$b \leftarrow c \text{ lsr } 1$	Logical shift right by one
0x1E	TRAP_S	Trap	Raise Exception
0x1F	BRK_S	Break	Break (Encoding is 0x7FFF)

9.8.2 SOP-format 16-bit General Operations

Figure 9-14 Compact Single Operand, Jumps, Special Formats

Syntax:

op_S b

op_S b,b

J_S<.d> [b]

JL_S<.d> [b]
 SUB_S.ne b,b,b

Table 9-6 16-Bit Single Operand Instructions

Sub- opcode2 c field (3 bits)	Instruction	Operation	Description
0x00	J_S	$pc \leftarrow b$	Jump
0x01	J_S.D	$pc \leftarrow b$	Jump delayed
0x02	JL_S	$blink \leftarrow next\_pc; pc \leftarrow b$	Jump and link
0x03	JL_S.D	$blink \leftarrow next\_pc; pc \leftarrow b$	Jump and link delayed
0x04		Illegal Instruction	Reserved
0x05		Illegal Instruction	Reserved
0x06	SUB_S.NE	if (flags.Z==0) then $b \leftarrow b - b$	If Z flag is 0, clear register
0x07	ZOP s	b field is sub-opcode3	See Table 9-7 on page 396.

9.8.3 ZOP-format 16-bit General Operations

Figure 9-15 Compact Zero Operand Instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					i[2:0]			0x07			0x00				
0	1	1	1	1	i	i	i	1	1	1	0	0	0	0	0

Syntax:

op_S

NOP_S

UNIMP_S

J_S<.d> [blink]

JEQ_S [blink]

JNE_S [blink]

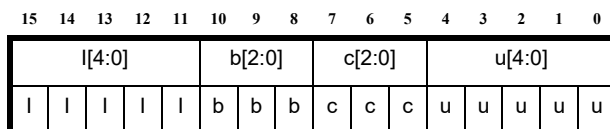
Table 9-7 16-Bit Zero Operand Instructions

Sub- opcode3 b field (3 bits)	Instruction	Operation	Description
0x00	NOP_S	nop	No operation
0x01	UNIMP_S	Illegal Instruction	Unimplemented Instruction
0x02	SWI_S	Software Interrupt	Raise a software interrupt exception.
0x03		Illegal Instruction	Reserved
0x04	JEQ_S [blink]	pc ← blink	Jump using blink register if equal
0x05	JNE_S [blink]	pc ← blink	Jump using blink register if not equal
0x06	J_S [blink]	pc ← blink	Jump using blink register
0x07	J_S.D [blink]	pc ← blink	Jump using blink register delayed

9.9 16-bit Load and Store Formats with Offset

The offset $u[4:0]$ is data size aligned. Syntactically, $u7$ must be multiples of 4, and $u6$ must be multiples of 2.

Figure 9-16 Compact Load/Store with Offset Format



Syntax:

LD_S c, [b, u7] (*u7 must be 32-bit aligned*)

LDB_S c, [b, u5]

LDH_S c, [b, u6] (*u6 must be 16-bit aligned*)

LDH_S.X c, [b, u6] (*u6 must be 16-bit aligned*)

ST_S c, [b, u7] (*u7 must be 32-bit aligned*)

STB_S c, [b, u5]

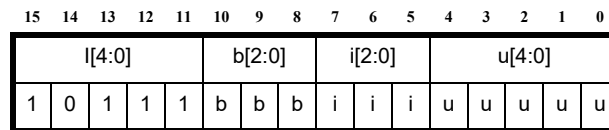
STH_S c, [b, u6] (*u6 must be 16-bit aligned*)

Table 9-8 Summary of 16-Bit Load and Store Instructions with Offset

Top-level format	Major opcode I[4:0]	Instruction	Operation	Description
F16_LD_WORD	0x10	LD_S	$c \leftarrow \text{mem}[b + u7].l$	Load 32-bit word
F16_LD_BYTE	0x11	LDB_S	$c \leftarrow \text{mem}[b + u5].b$	Load unsigned byte
F16_LD_HALF	0x12	LDH_S	$c \leftarrow \text{mem}[b + u6].w$	Load unsigned 16-bit half-word
F16_LDX_HALF	0x13	LDH_S.X	$c \leftarrow \text{mem}[b + u6].wx$	Load signed 16-bit half-word
F16_ST_WORD	0x14	ST_S	$\text{mem}[b + u7].l \leftarrow c$	Store 32-bit word
F16_ST_BYTE	0x15	STB_S	$\text{mem}[b + u5].b \leftarrow c$	Store unsigned byte
F16_ST_HALF	0x16	STH_S	$\text{mem}[b + u6].w \leftarrow c$	Store unsigned 16-bit half-word

9.10 Shift/Subtract/Bit Immediate [F16_SH_SUB_BIT]

Figure 9-17 Compact Shift/Sub Bit Immediate Format



Syntax:

SUB_S b, b, u5

BSET_S b, b, u5

BCLR_S b, b, u5

BMSK_S b, b, u5

BTST_S b, u5

ASL_S b, b, u5

LSR_S b, b, u5

ASR_S b, b, u5

Table 9-9 16-Bit Shift/SUB/Bit Immediate

Sub-opcode2 i field (3 bits)	Instruction	Operation	Description
0x00	ASL_S	$b \leftarrow b \text{ asl } u5$	Multiple arithmetic shift left
0x01	LSR_S	$b \leftarrow b \text{ lsr } u5$	Multiple logical shift right
0x02	ASR_S	$b \leftarrow b \text{ asr } u5$	Multiple arithmetic shift right
0x03	SUB_S	$b \leftarrow b - u5$	Subtract
0x04	BSET_S	$b \leftarrow b \text{ or } 1 \ll u5$	Bit set
0x05	BCLR_S	$b \leftarrow b \text{ and not } 1 \ll u5$	Bit clear
0x06	BMSK_S	$b \leftarrow b \text{ and } ((1 \ll (u5+1)) - 1)$	Bit mask
0x07	BTST_S	$b \text{ and } 1 \ll u5$	Bit test

9.11 Stack-based Operations [F16_SP_OPS]

Instruction group 0x18 contains a collection of 16-bit formats designed specifically to encode common stack-based operations in which the SP register (r28) is always an implied operand.

Within this format, the $i[2:0]$ field encodes sub-opcodes 0 to 7, selecting from the range of operands described in Table 9-10. Sub-opcodes 0 to 4 each encode a single unique operation (LD_S, LDB_S, ST_S, STB_S, and ADD_S). However, sub-opcodes 5, 6, and 7 each encode two or more operations by further encoding either the $b[2:0]$ field or the $u[4:0]$ field.

Table 9-10 Opcodes for Stack-based Operations

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Encoded Instructions		l[4:0]				b[2:0]			i[2:0]			u[4:0]					
LD_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	0	0	u	u	u	u	u
LDB_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	0	1	u	u	u	u	u
ST_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	1	0	u	u	u	u	u
STB_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	1	1	u	u	u	u	u
ADD_S	b, SP, u7	1	1	0	0	0	b	b	b	1	0	0	u	u	u	u	u
ADD_S	SP, SP, u7	1	1	0	0	0	0	0	0	1	0	1	u	u	u	u	u
SUB_S	SP, SP, u7	1	1	0	0	0	0	0	1	1	0	1	u	u	u	u	u
POP_S	b	1	1	0	0	0	b	b	b	1	1	0	0	0	0	0	1
POP_S	BLINK	1	1	0	0	0	R	R	R	1	1	0	1	0	0	0	1
PUSH_S	b	1	1	0	0	0	b	b	b	1	1	1	0	0	0	0	1
PUSH_S	BLINK	1	1	0	0	0	R	R	R	1	1	1	1	0	0	0	1
LEAVE_S	u7	1	1	0	0	0	U	U	U	1	1	0	u	u	u	u	0
ENTER_S	u6	1	1	0	0	0	0	U	U	1	1	1	u	u	u	u	0

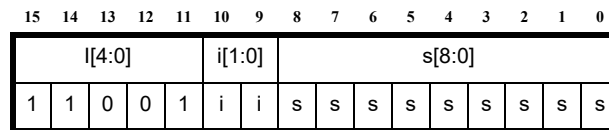
The ENTER_S and LEAVE_S instructions are available only when the code_density extension pack is configured in the architecture. If an ENTER_S or LEAVE_S instruction is executed on a build without the code_density extension pack, an Illegal Instruction Error exception is raised.

This format contains a number of unused encodings. There are, for example, 6 unused encodings in sub-opcode 5, for values of $b[2:0]$ in the range 2 to 7. Any attempt to execute an unused encoding raises an Illegal Instruction Error exception.

Following the normal rule for the use of reserved fields, any non-zero value in a reserved field is ignored. However, reserved bits must be set to 0 when assembling instructions.

9.12 Load/Add GP-Relative [F16_GP_LD_ADD]

Figure 9-18 Compact Load/Add GP-Relative Format



The offset (s[8:0]) is shifted accordingly to provide the appropriate data size alignment.

Syntax:

LD_S r0, [GP, s11] *(32-bit aligned offset)*

LDB_S r0, [GP, s9] *(8-bit aligned offset)*

LDH_S r0, [GP, s10] *(16-bit aligned offset)*

ADD_S r0, GP, s11 *(32-bit aligned offset)*

Table 9-11 16-Bit GP Relative Instructions

Sub- opcode i field (2 bits)	Instruction	Operation	Description
0x00	LD_S	$r0 \leftarrow \text{mem}[GP + s11].l$	Load GP-relative (32-bit aligned) to r0
0x01	LDB_S	$r0 \leftarrow \text{mem}[GP + s9].b$	Load unsigned byte GP-relative (8-bit aligned) to r0
0x02	LDH_S	$r0 \leftarrow \text{mem}[GP + s10].w$	Load unsigned 16-bit half-word GP-relative (16-bit aligned) to r0
0x03	ADD_S	$r0 \leftarrow GP + s11$	Add GP-relative (32-bit aligned) to r0

9.13 Load PCL-Relative [F16_PCL_LD]

The offset (u[7:0]) is shifted accordingly to provide the appropriate 32-bit data size alignment.

Figure 9-19 Compact Load PCL Relative Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
l[4:0]					b[2:0]			u[7:0]							
1	1	0	1	0	b	b	b	u	u	u	u	u	u	u	u

Syntax:

```
LD_S b, [PCL, u10] (32-bit aligned offset)
```

9.14 Move Immediate [F16_MV_IMM]

Figure 9-20 Compact Move Immediate Format

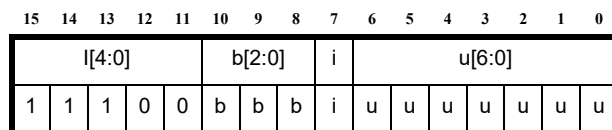
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					b[2:0]			u[7:0]							
1	1	0	1	1	b	b	b	u	u	u	u	u	u	u	u

Syntax:

MOV_S b, u8

9.15 ADD/CMP Immediate [F16_OP_IMM]

Figure 9-21 Compact ADD/CMP Immediate Format



Syntax:

ADD_S b, b, u7

CMP_S b, u7

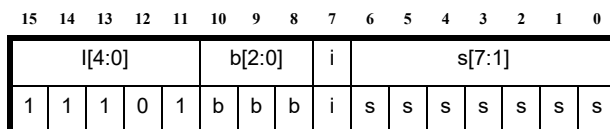
Table 9-12 16-Bit ADD/CMP Immediate

Sub- opcode i field (1 bit)	Instruction	Operation	Description
0x00	ADD_S	$b \leftarrow b + u7$	Add
0x01	CMP_S	$b - u7$	Compare

9.16 Branch on Compare Register with Zero [F16_BCC_REG]

The target address is 16-bit aligned.

Figure 9-22 Compact Branch on Compare Register with Zero Format



Syntax:

BREQ_S b, 0, s8

BRNE_S b, 0, s8

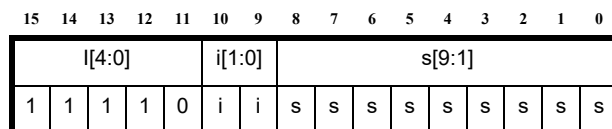
Table 9-13 16-Bit Branch on Compare

Sub-opcode i field (1 bit)	Instruction	Operation	Description
0x00	BREQ_S	if (b==0) PC = (PCL +s8)	Branch if register is zero
0x01	BRNE_S	if (b!=0) PC = (PCL +s8)	Branch if register is non-zero

9.17 Branch Conditionally [F16_BCC]

The target address is 16-bit aligned.

Figure 9-23 Compact Branch Conditionally Format



Syntax:

B_S s10

BEQ_S s10

BNE_S s10

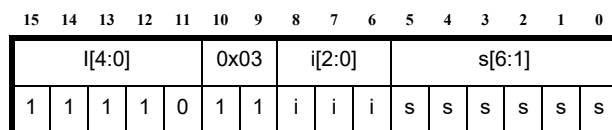
Table 9-14 16-Bit Branch, Branch Conditionally

Sub- opcode i field (2 bits)	Instruction	Operation	Description
0x00	B_S	PC = (PCL+s10)	Branch always
0x01	BEQ_S	if (Z) PC = (PCL+s10)	Branch if equal
0x02	BNE_S	if (/Z) PC = (PCL+s10)	Branch if not equal
0x03	Bcc_S	if (cc) PC = (PCL+s10)	See Bcc_S

9.17.1 Branch Conditionally with cc Field, 0x1E, [0x03, 0x00 – 0x07]

The target address is 16-bit aligned.

Figure 9-24 Branch Conditionally with cc Field Format



Syntax:

BGT_S s7

BGE_S s7

BLT_S s7

BLE_S s7

BHI_S s7

BHS_S s7

BLO_S s7

BLS_S s7

Table 9-15 16-Bit Branch Conditionally

Sub- opcode i field (3 bits)	Instruction	Operation	Description
0x00	BGT_S	if ((N and V and /Z) or (/N and /V and /Z)) PC = (PCL+s7)	Branch if greater than
0x01	BGE_S	if ((N and V) or (/N and /V)) PC = (PCL+s7)	Branch if greater than or equal
0x02	BLT_S	if ((N and /V) or (/N and V)) PC = (PCL+s7)	Branch if less than
0x03	BLE_S	if (Z or (N and /V) or (/N and V)) PC = (PCL+s7)	Branch if less than or equal
0x04	BHI_S	if (/C and /Z) PC = (PCL+s7)	Branch if higher than
0x05	BHS_S	if (/C) PC = (PCL+s7)	Branch if higher or the same
0x06	BLO_S	if (C) PC = (PCL+s7)	Branch if lower than
0x07	BLS_S	if (C or Z) PC = (PCL+s7)	Branch if lower or the same



Note

For details on the codes used for condition code tests, see [Table 7-7](#).

9.18 Branch and Link Unconditionally [F16_BL]

The target address can only target 32-bit aligned instructions.

Figure 9-25 Compact Branch and Link Unconditionally Format



Syntax:

BL_S s13

10

Instruction Set Details

This chapter lists the available instruction set in alphabetic order. The syntax and encoding examples list full syntax for each instruction, but excludes the redundant encoding formats. A full list of encoding formats can be found in [“Instruction Set Summary”](#).

Both 32-bit and 16-bit instruction encodings are available in the ARCV2 ISA and are indicated using particular suffixes on the instruction as illustrated by the following syntax:

OP implies 32-bit encoding

OP_L indicates 32-bit encoding.

OP_S indicates 16-bit encoding

If no suffix is used on the instruction then the implied instruction format is 32 bits.

[Table 7-16](#) on page 314 lists the syntax conventions. [Table 7-5](#) on page 306 and [Table 7-6](#) on page 307 lists the encoding notation.

[Table 10-1](#) summarizes the 32-bit format instructions alongside the 16-bit format instructions supported by the ARCV2 ISA.

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [Dual-operand Instructions, F32_GEN4](#).

10.1 Instruction List

Table 10-1 List of Instructions

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
ABS	Absolute value	ABS_S	Absolute value
ADC	Add with carry		
ADD	Add	ADD_S	Add
ADD1	Add with left shift by 1 bit	ADD1_S	Add with left shift by 1 bits
ADD2	Add with left shift by 2 bits	ADD2_S (see ADD2)	Add with left shift by 2 bits
ADD3	Add with left shift by 3 bits	ADD3_S (see ADD3)	Add with left shift by 3 bits
AEX	Swap contents of auxiliary register with a 32-bit core register		
AND	Logical AND	AND_S (see AND)	Logical AND
ASL	Arithmetic Shift Left	ASL_S (see ASL)	Arithmetic Shift Left
ASR	Arithmetic Shift Right	ASR_S (see ASR)	Arithmetic Shift Right
ASR16	Arithmetic Shift Right by 16		
ASR8	Arithmetic Shift Right by 8		
B	Branch unconditionally	B_S	Branch unconditionally
BBIT0	Branch if bit equal to 0		
BBIT1	Branch if bit equal to 1		
Bcc	Branch if condition true	Bcc_S	Branch if condition true
BCLR	Clear specified bit (to 0)	BCLR_S (see BCLR)	Clear specified bit (to 0)
BI	Branch Indexed, 32-bit full-word table		
BIH	Branch Indexed, 16-bit Half-word table		
BIC	Bit-wise inverted AND	BIC_S (see BIC)	Bit-wise inverted AND
BLcc	Branch and Link	BL_S (see BLcc)	Branch and Link

Table 10-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
BMSK	Bit Mask	BMSK_S (see BMSK)	Bit Mask
BMSKN	Bit Mask Negated		
BRcc	Branch on compare	BRcc_S (see BRcc)	Branch on compare
BRK	Break (halt) processor	BRK_S (see BRK)	Break (halt) processor
BSET	Set specified bit (to 1)	BSET_S (see BSET)	Set specified bit (to 1)
BSPEEK	Peek the head of the bitstream		
BSPOP	Pop head of the bitstream		
BSPUSH	Push to the tail of the bitstream		
BTST	Test value of specified bit	BTST_S (see BTST)	Test value of specified bit
BXOR	Bit XOR		
CLRI	Clear Interrupt Enable		
CMP	Compare	CMP_S (see CMP)	Compare
DBNZ	Signed integer Divide		
DADDH11	Add to D1, writing to D1		
DADDH12	Add to D2, writing to D1		
DADDH21	Add to D1, writing to D2		
DADDH22	Add to D2, writing to D2		
DEXCL1	Register D1 Exchange		
DEXCL2	Register D2 Exchange		
DMULH11	Multiply by D1, writing to D1		
DMULH12	Multiply by D2, writing to D1		
DMULH21	Multiply by D1, writing to D2		
DMULH22	Multiply by D2, writing to D2		
DSUBH11	Subtract from D1, writing to D1		
DSUBH12	Subtract from D2, writing to D1		
DSUBH21	Subtract from D1, writing to D2		
DSUBH22	Subtract from D2, writing to D2		

Table 10-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
DIV	Signed integer Divide		
DIVU	Unsigned integer Divide		
		EI_S	Execute Indexed
		ENTER_S	Function prolog Sequence
EX	Atomic Exchange		
EXTB	Zero-extend byte to word	EXTB_S (see EXTB)	Zero-extend byte
EXTH	Zero-extend 16-bit half-word to word	EXTH_S (see EXTH)	Zero-extend 16-bit half-word
FCVT32	Conversion between 32-bit data formats		
FFS	Find First Set		
FLAG	Write to Status Register		
FLS	Find Last Set		
FSABS	Single-precision floating-point absolute operation		
FSADD	Single-precision floating-point addition		
FSCMP	Single-precision floating-point comparison		
FSCMPF	Single-precision floating-point comparison		
FSDIV	Single-precision floating-point division		
FSMADD	Single-precision floating-point fusedMultiplyAdd A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding.		

Table 10-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
FSMSUB	Single-precision floating-point fusedMultiplySubtract A fusedMultiplySubtract is a floating-point multiply-subtract operation performed in one step, with a single rounding.		
FSMUL	Single-precision floating-point multiplication		
FSNEG	Single-precision floating-point negation		
FSSQRT	Single-precision floating-point square-root		
FSSUB	Single-precision floating-point subtraction		
Jcc	Jump	J_S (see Jcc)	Jump
JL	Jump	JL_S (see Jcc)	Jump and Link
		JLI_S	Jump and Link Indexed
KFLAG	Write to Status Register in kernel mode		
LD LDH LDW LDB	Load from memory	LD_S (see LD LDH LDW LDB)	Load from memory
LDI (see LDI, LDI_S)	Load Indexed	LDI_S (see LDI, LDI_S)	Load Indexed
		LEAVE_S	Function Epilog Sequence
LP<cc>	Loop (zero-overhead loops)		
LR	Load from Auxiliary memory		
LSL16	Logical Shift Left 16		
LSL8	Logical Shift Left 8		
LSR	Logical Shift Right	LSR_S (see LSR)	Logical Shift Right
LSR16	Logical Shift Right 16		
LSR8	Logical Shift Right 8		
MAX	Return Maximum		

Table 10-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
MIN	Return Minimum		
MOV	Move (copy) to register	MOV_S (see MOV)	Move (copy) to register
MPY, MPY_S	32 x 32 Signed Multiply (lsw)	MPY_S (see MPY, MPY_S)	32 x 32 Signed Multiply (lsw)
MPYM MPYH	32 x 32 Signed Multiply (msw)		
MPYMU MPYHU	32 x 32 Unsigned Multiply (msw)		
MPYU	32 x 32 Unsigned Multiply (lsw)		
MPYUW, MPYUW_S	16 bit unsigned integer multiplication	MPYUW_S (see MPYUW, MPYUW_S)	
MPYW, MPYW_S	16 X 16 signed multiply	MPYW_S (see MPYW, MPYW_S)	16 x16 signed multiply
NEG	Negate	NEG_S (see NEG)	Negate
NEXTTASK	Switch the task and the register bank		
NOP	No operation	NOP_S (see NOP)	No operation
NORM	Normalize to 32 bits		
NORMH NORMW	Normalize to 16 bits		
NOT	Logical bit inversion	NOT_S (see NOT)	Logical bit inversion
OR	Logical OR	OR_S (see OR)	Logical OR
		POP_S	Restore register from stack
PREFETCH	Prefetch from memory		
		PUSH_S	Store register to the stack
RCMP	Reverse Compare		
REM	Signed Integer Remainder		
REMU	Unsigned Integer Remainder		
RLC	Rotate Left through Carry		
ROR	Rotate Right		

Table 10-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
ROR8	Rotate Right 8		
ROL	Rotate Left		
ROL8	Rotate Left 8		
RRC	Rotate Right through Carry		
RSUB	Reverse Subtraction		
RTIE	Return from Interrupt or Exception		
SBC	Subtract with carry		
SETcc	Set conditional		
SETI	Set Interrupt Enable		
SEXB	Sign-extend byte to word	SEXB_S (see SEXB)	Sign-extend byte
SEXH SEXW	Sign-extend half-word to word	SEXH_S (see SEXH SEXW)	Sign-extend 16-bit half-word
SFLAG	Set secure status flags		
SJLI	Secure jump and link indirect		
SLEEP	Put processor in sleep state		
SR	Store to Auxiliary memory		
ST ST STH STB	Store to memory	ST_S (see ST ST STH STB)	Store to memory
SUB	Subtract	SUB_S (see SUB)	Subtract
SUB1	Subtract with left shift by 1 bit		
SUB2	Subtract with left shift by 2 bits		
SUB3	Subtract with left shift by 3 bits		
SWAP	Swap 16-bit register halves		
SWAPE	Swap byte ordering		
SWI	Software interrupt	SWI_S (see SWI)	Software interrupt
SYNC	Synchronize		
		TRAP_S	Trap to system call
TST	Test	TST_S (see TST)	Test

Table 10-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
WEVT	Enter sleep state and wait on event		
XBFU	Extract unsigned bit-field		
XOR	Logical Exclusive-OR	XOR_S (see XOR)	Logical Exclusive-OR
		UNIMP_S	Unimplemented Instruction

10.2 ARC Instructions

This section provides a complete definition for each ARCV2 instruction, presented in alphabetical order. The instruction name is given at the top left and top right of the page, along with a brief instruction description, and instruction type.

The following sub-headings are used in the description of each instruction.

Function	Summarizes the function of the instruction.
Extension Group	Indicates if instruction is in the baseline set or included only as part of an extension group
Operation	C style expression that describes the operation of the instruction. Where relevant, a block diagram of the computation may be shown.
Instruction Format	Summary of the instruction format.
Syntax Example	Single syntax example.
Flag Affected	List of status flags that are affected.
Description	Full description of the instruction.
Pseudo Code	Operation of the instruction described in C style pseudo code.
Assembly Code Example	Assembly coding example.
Syntax and Encoding	The syntax of the instruction and corresponding instruction encoding. Table 7-16 on page 314 lists the instruction syntax convention. Table 7-5 on page 306 and Table 7-6 on page 307 lists the key for encoding conventions.

ABS

Function

Absolute

Extension Group

BASELINE

Operation

$b = \text{ABS}(c);$

Instruction Format

op b, c

Syntax Example

$\text{ABS}<.\text{f}> \text{ b}, \text{ c}$

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if src = 0x8000 0000
C	•	= MSB of src
V	•	= Set if src = 0x8000 0000

Description

Take the absolute value that is found in the source operand (c) and place the result into the destination register (b). The carry flag reflects the state of the most-significant bit found in the source register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

/* ABS */
alu = 0 - src
if src[31]==1 then
    dest = alu
else
    dest = src
if F==1 then
    STATUS32[Z] = if dest==0 then 1 else 0
    STATUS32[N] = if src==0x8000_0000 then 1 else 0
    STATUS32[C] = src[31]
    STATUS32[V] = if src==0x8000_0000 then 1 else 0

```

Assembly Code Example

```
ABS r1,r2      ; Take the absolute value of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ABS<.f>	b, c	00100bbb00101111FBBBCCCCC001001
ABS<.f>	b, u6	00100bbb01101111FBBBuuuuuu001001
ABS_S	b, c	01111bbbccc10001

ADC

Function

Addition with Carry

Extension Group

BASELINE

Operation

if (cc) $a = b + c + \text{carry}$

Instruction Format

op a, b, c

Syntax Example

ADC<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Add source operand 1 (b) and source operand 2 (c) and carry, and place the result in the destination register, a. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* ADC */
    dest = src1 + src2 + C_flag
    if F==1 then
        Z_flag = if dest==0 then 1 else
0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
ADC r1,r2,r3    ; Add r2 to r3 with carry and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADC<.f>	a,b,c	00100bbb00000001FBBBCCCCC0AAAAAA
ADC<.f>	a,b,u6	00100bbb010000001FBBBuuuuuu0AAAAAA
ADC<.f>	b,b,s12	00100bbb10000001FBBBssssss0SSSSSS
ADC<.cc><.f>	b,b,c	00100bbb110000001FBBBCCCCC00QQQQQ
ADC<.cc><.f>	b,b,u6	00100bbb110000001FBBBuuuuuu10QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ADD

Extension Group

BASELINE

Operation

if (cc) $a = b + c$

Instruction Format

op a, b, c

Syntax Example

ADD<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Add source operand 1 (b) to source operand 2 (c) and place the result in the destination register, a. Any flag updates occur only if the set flags suffix (.F) is used.



Note

For 16-bit encoded instructions that work on the stack pointer (SP) or global pointer (GP), the offset is aligned to 32-bit. For example, ADD_S sp, sp, u7 only needs to encode the top 5 bits because the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

Pseudo Code

```

if cc==true then                                /* ADD */
    dest = src1 + src2
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
ADD r1,r2,r3    ; Add contents of r2 with r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADD<.f>	a,b,c	00100bbb0000000FBBBCCCCCAAAAAA
ADD<.f>	a,b,u6	00100bbb01000000FBBBuuuuuuAAAAAA
ADD<.f>	b,b,s12	00100bbb1000000FBBBssssssSSSSSS
ADD<.cc><.f>	b,b,c	00100bbb11000000FBBBCCCCC0QQQQQ
ADD<.cc><.f>	b,b,u6	00100bbb11000000FBBBuuuuuu1QQQQQ
ADD_S	a,b,c	01100bbbccc11aaa
ADD_S	b,b,h	01110bbbhhh000HH
ADD_S	h,h,s3	01110ssshhh001HH
ADD_S	b,b,limm	01110bbb11000011
ADD_S	0,limm,s3	01110sss11000111
ADD_S	b,sp,u7	11000bbb100uuuuu
ADD_S	b,b,u7	11100bbb0uuuuuuu
ADD_S	c,b,u3	01101bbbccc00uuu
ADD_S	SP,SP,u7	11000000101uuuuu
ADD_S	R0,GP,s11	1100111sssssssss

Encodings supported by the CODE DENSITY option

ADD_S	R0,b,u6	01001bbb0UUU1uuu
ADD_S	R1,b,u6	01001bbb1UUU1uuu

ADD1

Function

Addition with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b + (c \ll 1)$;

Instruction Format

op a, b, c

Syntax Example

ADD1<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 1. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD1 */
if cc==true then
  shiftedsrc2 = src2 << 1
  dest = src1 + shiftedsrc2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
    ( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31] )
```


Assembly Code Example

```
ADD1 r1,r2,r3    ; Add contents of r3 shifted left one bit to r2 and
                  ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADD1<.f>	a,b,c	00100bbb00010100FBBBCCCCCAAAAAA
ADD1<.f>	a,b,u6	00100bbb01010100FBBBuuuuuuAAAAAA
ADD1<.f>	b,b,s12	00100bbb10010100FBBBssssssSSSSSS
ADD1<.cc><.f>	b,b,c	00100bbb11010100FBBBCCCCC0QQQQQ
ADD1<.cc><.f>	b,b,u6	00100bbb11010100FBBBuuuuuu1QQQQQ
ADD1_S	b,b,c	01111bbbccc10100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ADD2

Function

Addition with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b + (c \ll 2)$;

Instruction Format

op a, b, c

Syntax Example

ADD2<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 2. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD2 */
if cc==true then
  shiftedsrc2 = (src2 << 2)
  dest = src1 + shiftedsrc2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
    ( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31] )
```

Assembly Code Example

```
ADD2 r1,r2,r3      ; Add contents of r3 shifted left two bits to r2 and
                   ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADD2<.f>	a,b,c	00100bbb00010101FBBBCCCCC AAAAAA
ADD2<.f>	a,b,u6	00100bbb01010101FBBBuuuuuu AAAAAA
ADD2<.f>	b,b,s12	00100bbb10010101FBBBssssss SSSSSS
ADD2<.cc><.f>	b,b,c	00100bbb11010101FBBBCCCCC 0QQQQQ
ADD2<.cc><.f>	b,b,u6	00100bbb11010101FBBBuuuuuu 1QQQQQ
ADD2_S	b,b,c	01111bbbccc10101

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ADD3

Function

Addition with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b + (c \ll 3)$;

Instruction Format

op a, b, c

Syntax Example

ADD3<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 3. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD3 */
if cc==true then
  shiftedsrc2 = src2 << 3
  dest = src1 + shiftedsrc2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
    ( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31] )
```

Assembly Code Example

```
ADD3 r1,r2,r3    ; Add contents of r3 shifted left three bits to r2 and
                  ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADD3<.f>	a,b,c	00100bbb00010110FBBBCCCCCAAAAAA
ADD3<.f>	a,b,u6	00100bbb01010110FBBBuuuuuuAAAAAA
ADD3<.f>	b,b,s12	00100bbb10010110FBBBssssssSSSSSS
ADD3<.cc><.f>	b,b,c	00100bbb11010110FBBBCCCCC0QQQQQ
ADD3<.cc><.f>	b,b,u6	00100bbb11010110FBBBuuuuuu1QQQQQ
ADD3_S	b,b,c	01111bbbccc10110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

AEX

Function

Swap contents of an auxiliary register with a core register.

Extension Group

BASELINE

Instruction Format

op b, [c]

Syntax Example

AEX<.cc> b, [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Swaps contents of an auxiliary register with a core register. This instruction is used by the micro-operations sequencer within the exception/interrupt entry/exit sequences. The b register operand of AEX cannot specify either a LImm data value or a null operand. Any use of LImm or null operand for the b register raises an [Illegal Instruction](#) exception.

Semantically, an AEX instruction behaves as the union of the LR and SR instructions, with the exception that an AEX can be conditional whereas LR/SR are not.

Pseudo Code

```
if cc==true then                                /* AEX */
    tmp = AuxRead (src2)
    AuxWrite (src2, b)
    b = tmp
```

Assembly Code Example

```
AEX R28, [AUX_USER_SP]    ; swap user and kernel stack pointers
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
AEX	b, [c]	00100bbb00100111RBBBCCCCCRRRRRR
AEX	b, [u6]	00100bbb01100111RBBBuuuuuuRRRRRR
AEX	b, [s12]	00100bbb10100111RBBBssssssSSSSSS
AEX<.cc>	b, [c]	00100bbb11100111RBBBCCCCC0QQQQQ
AEX<.cc>	b, [u6]	00100bbb11100111RBBBuuuuuu1QQQQQ

AND

Function

Bitwise AND Operation

Extension Group

BASELINE

Operation

if (cc) a = b & c;

Instruction Format

op a, b, c

Syntax Example

AND<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise AND of source operand 1 (b) with source operand 2 (c) with the result written to the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                /* AND */
  dest = src1 AND src2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
AND r1,r2,r3    ; AND contents of r2 with r3 and write result into r1
```


Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
AND<.f>	a, b, c	00100bbb00000100FBBBCCCCCAAAAAA
AND<.f>	a, b, u6	00100bbb01000100FBBBuuuuuuAAAAAA
AND<.f>	b, b, s12	00100bbb10000100FBBBssssssSSSSSS
AND<.cc><.f>	b, b, c	00100bbb11000100FBBBCCCCC0QQQQQ
AND<.cc><.f>	b, b, u6	00100bbb11000100FBBBuuuuuu1QQQQQ
AND_S	b, b, c	01111bbbccc00100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ASL

Function

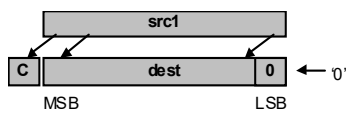
Arithmetic Shift Left

Extension Group

BASELINE

Operation

$b = (\text{signed})\ c \ll 1;$



Instruction Format

op b, c

Syntax Example

ASL<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if the sign bit changes after a shift

Description

Arithmetically, left shift the source operand (c) by one and place the result into the destination register (b). An ASL operation is effectively accomplished by adding the source operand upon itself ($c + c$), with the result being written into the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src + src                                /* ASL */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
ASL r1,r2    ; Arithmetic shift left contents of r2 by one bit and
              ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASL<.f>	b,c	00100bbb00101111FBBBCCCCC000000
ASL<.f>	b,u6	00100bbb01101111FBBBuuuuuu000000
ASL_S	b,c	01111bbbccc11011

ASL Multiple

Function

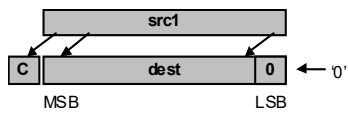
Multiple Arithmetic Shift Left

Extension Group

SHIFT_OPTION 2 or 3

Operation

if (cc) a = (signed) b << c;



Instruction Format

op a, b, c

Syntax Example

ASL<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Arithmetically, shift left b by c places and place the result in the destination register. Only the bottom 5 bits of c are used as the shift value. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* ASL */
    dest = src1 << (src2 & 31)                  /* Multiple */
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = if src2==0 then 0 else src1[32-src2]

```

Assembly Code Example

```

ASL r1,r2,r3    ; Arithmetic shift left contents of r2 by r3 bits
                ; and write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASL<.f>	a,b,c	00101bbb0000000FBBBCCCCCAAAAAA
ASL<.f>	a,b,u6	00101bbb01000000FBBBuuuuuuAAAAAA
ASL<.f>	b,b,s12	00101bbb1000000FBBBssssssSSSSSS
ASL<.cc><.f>	b,b,c	00101bbb11000000FBBBCCCCC0QQQQQ
ASL<.cc><.f>	b,b,u6	00101bbb11000000FBBBuuuuuu1QQQQQ
ASL_S	c,b,u3	01101bbbccc10uuu
ASL_S	b,b,c	01111bbbccc11000
ASL_S	b,b,u5	10111bbb000uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ASR

Function

Arithmetic Shift Right

Extension Group

BASELINE

Operation

$$b = (\text{signed})\ c \gg 1;$$



Instruction Format

op b, c

Syntax Example

ASR<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Arithmetically right shift the source operand (c) by one and place the result into the destination register (b). The sign of the source operand is retained in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

dest = src >> 1                                /* ASR */
if src[31]==1 then dest[31] = 1
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[0]

```

Assembly Code Example

```

ASR r1,r2      ; Arithmetic shift right contents of r2 by one bit and
                ;write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASR<.f>	b,c	00100bbb00101111FBBBCCCCC000001
ASR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000001
ASR_S	b,c	01111bbbccc11100

ASR multiple

Function

Multiple Arithmetic Shift Right

Extension Group

SHIFT_OPTION 2 or 3

Operation

if (cc) a = (signed) b >> c;



Instruction Format

op a, b, c

Syntax Example

ASR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Arithmetically, shift right b by c places and place the result in the destination register. Only the bottom 5 bits of c are used as the shift value. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* ASR */
  dest = ((signed)src1) >> (src2 & 31)              /* Multiple */
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = if src2==0 then 0 else src1[src2-1]

```

Assembly Code Example

```

ASR r1,r2,r3    ; Arithmetic shift right contents of r2 by r3 bits
                ; and write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASR<.f>	a,b,c	00101bbb00000010FBBBCCCCCAAAAAA
ASR<.f>	a,b,u6	00101bbb01000010FBBBuuuuuuAAAAAA
ASR<.f>	b,b,s12	00101bbb10000010FBBBssssssSSSSSS
ASR<.cc><.f>	b,b,c	00101bbb11000010FBBBCCCCC0QQQQQ
ASR<.cc><.f>	b,b,u6	00101bbb11000010FBBBuuuuuu1QQQQQ
ASR_S	c,b,u3	01101bbbccc11uuu
ASR_S	b,b,c	01111bbbccc11010
ASR_S	b,b,u5	10111bbb010uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ASR16

Function

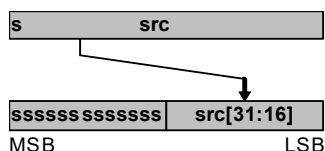
Arithmetic Shift Right 16

Extension Group

SHIFT_OPTION 1 or 3

Operation

$b = c \gg 16;$



Instruction Format

op b,c

Syntax Example

ASR16<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 16 places to the right. Set the upper 16 bits to 0xFFFF if the source operand is negative. Otherwise, clear the upper 16 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 16                                /* ASR16 */
if F==1 then
    Z_flag = if dest==0 then 1 else
0
    N_flag = dest[31]
```

Assembly Code Example

```
ASR16 r1,r2    ; Arithmetic shift of r2 16 places to the
                ; right, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASR16<.f>	b,c	00101bbb00101111FBBBCCCCC001100
ASR16<.f>	b,u6	00101bbb01101111FBBBuuuuuu001100

ASR8

Function

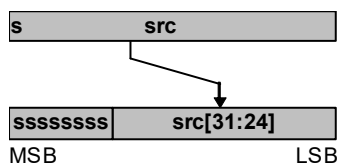
Arithmetic Shift Right 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$b = c \gg 8;$



Instruction Format

op b,c

Syntax Example

ASR8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 8 places to the right. Set the upper 8 bits to 0xFF if the source operand is negative. Otherwise, clear the upper 8 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 8                                /* ASR8 */
if F==1 then
    Z_flag = if dest==0 then 1 else
0
    N_flag = dest[31]
```

Assembly Code Example

```
ASR8 r1,r2    ; Arithmetic shift of r2 8 places to the right, placing
               result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASR8<.f>	b,c	00101bbb00101111FBBBCCCCC001101
ASR8<.f>	b,u6	00101bbb01101111FBBBuuuuuu001101

B

Function

Branch unconditionally

Extension Group

BASELINE

Operation

$PC = (PCL + s25);$

Instruction Format

op s25

Syntax Example

B<.d> s25

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

When an unconditional branch is used, the program execution is resumed at location PC (actually PCL) + relative displacement, where PC is the address of the B instruction. The unconditional branch far format has a maximum branch range of +/- 16 M. Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction. The status flags are not updated with this instruction.

[Table 7-36](#) on page 332 describes the delay slot modes, .d.

[Table 7-7](#) on page 308 describes the condition codes, cc.



Caution

The B instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```
if N=1 then                                /* B */
    DelaySlot(nPC)
    PC = cPC + rd
```

Assembly Code Example

```
B label          ; Branch to label
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code

```
B<.d>          s25  00000ssssssssss1ssssssssssNRtttt
```

B_S

Function

16-Bit Branch

Extension Group

BASELINE

Operation

$PC = (PCL + s10)$

Instruction Format

op s10

Syntax Example

B_S s10

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

When using the B_S instruction, a branch is always executed from the current PC, 32-bit aligned, with the displacement value specified in the source operand.

For all branch types, the branch target is 16-bit aligned. The status flags are not updated with this instruction.



Caution

The B_S instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

Pseudo Code

```
KillDelaySlot(nPC)                                /* B_S */
PC = cPCL + rd
```

Assembly Code Example

```
B_S label ; Branch to label
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
B_S      s10  1111000ssssssss
```

BBIT0

Function

Branch on Bit Test Clear

Extension Group

BASELINE

Operation

if $((b \text{ AND } (1 \ll c)) == 0)$ PC = PCL+s9;

Instruction Format

op b, c, s9

Syntax Example

BBIT0<.d><.T> b, c, s9

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Test a bit within source operand 1 (b) to see if the bit is clear (0). Source operand 2 (c) explicitly specifies the position of the bit that is to be tested within source operand 1 (b). Only the bottom 5 bits of operand 2 are used as the bit position. If the bit is clear, the branch is taken. The branch target is computed as the sum of the current 32-bit word-aligned PC and the signed half-word displacement value specified by the 9-bit literal source operand (s9).

Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction. The status flags are not updated by this instruction.

[Table 7-36](#) on page [332](#) lists the delay slot modes, <.d>.



Caution

- The BBIT0 instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction.
- A BBIT0 instruction with a delay slot must not include a long immediate operand. If included, the processor raises an Illegal Instruction error.

Some implementations of the ARCV2 may predict the outcome of BBIT0 instructions. These predictions may be static, that is, based on the instruction pattern, whereas others may be dynamic, that is, based on the observed execution history of the branch. A description of the static prediction semantics for BBIT0 instructions can be found in section [Table 7-9](#) on page 310, and more details can be found in the section [“Branch On Compare or Bit Test”](#).

Pseudo Code

```
if (src1 & (1 << (src2 & 31)))==0    /* BBIT0 */
then
  if N=1 then
    DelaySlot(nPC)
    KillDelaySlot(nPC + 1)
    PC = PCL + s9
  else
    PC = nPC
```

Assembly Code Example

```
BBIT0 r1,9,label      ; Branch to label if bit 9
                       ; of r1 is clear
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BBIT0<.d><.T>	b,c,s9	00001bbbssssssss1SBBBCCCCCN0Y110
BBIT0<.d><.T>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN1Y110

BBIT1

Function

Branch on Bit Test Set

Extension Group

BASELINE

Operation

if ((b AND (1<<c)) == 1) PC = PCL+s9;

Instruction Format

op b, c, s9

Syntax Example

BBIT1<.d><.T> b, c, s9

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Test a bit within source operand 1 (b) to see if the bit is set (1). Source operand 2 (c) explicitly specifies the position of the bit that is to be tested within source operand 1 (b).

Only the bottom 5 bits of operand 2 are used as the bit position.

If the selected bit is set, the branch is taken. The branch target is computed as the sum of the current 32-bit word-aligned PC and the signed 16-bit half-word displacement value specified by the 9-bit literal source operand (s9).

Because the execution of the instruction that is in the delay slot is controlled by the delay slot mode, the instruction must never be the target of any branch or jump instruction. The status flags are not updated by this instruction.

[Table 7-36](#) on page [332](#) lists the delay slot modes, <.d>.

**Caution**

- The BBIT1 instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction.
- A BBIT1 instruction with a delay slot must not include a long immediate operand. If included, the processor raises an Illegal Instruction error.

Some implementations of the ARCV2 may predict the outcome of BBIT1 instructions. These predictions may be static, that is, based on the instruction pattern, whereas others may be dynamic, that is, based on the observed execution history of the branch. A description of the static prediction semantics for BBIT instructions can be found in section [“Static Branch Prediction Mode”](#), and more details can be found in section [“Branch On Compare or Bit Test”](#).

Pseudo Code

```
if (src1 & (1 << (src2 & 31))) != 0 then          /* BBIT1 */
  if N=1 then
    DelaySlot(nPC)
    KillDelaySlot(dPC)
    PC = cPCL + rd
  else
    PC = nPC
```

Assembly Code Example

```
BBIT1 r1,9,label    ; Branch to label if bit 9 of r1 is set
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BBIT1<.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCN0Y111
BBIT1<.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuuN1Y111

Bcc

Function

Branch Conditionally

Extension Group

BASELINE

Operation

if (cc) PC = (PCL+s21);

Instruction Format

op s21

Syntax Example

B<cc><.d> s21

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

When a conditional branch is used and the specified condition is met (cc = true), program execution is resumed at location PC (actually PCL) + relative displacement, where PC is the address of the Bcc instruction. The conditional branch instruction has a maximum range of +/- 1 M, and the target address is 16-bit aligned.

The unconditional branch far format has a maximum branch range of +/- 16 M. Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction. The status flags are not updated with this instruction.

The delay slot modes, .d, are described in [Table 7-36](#) on page 332.

The condition codes, cc, are described in [Table 7-7](#) on page 308.



Caution

The Bcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```
if cc==true then                                /* B */
  if N=1 then
    DelaySlot(nPC)
    PC = cPC + rd
  else
    PC = nPC
```

Assembly Code Example

```
BEQ label      ; Branch to label if Z flag is set Branch to label and
                ; execute the instruction in the delay ; slot if N flag
BPL.D label    ; is clear
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
B<cc><.d>    s21    00000 ssssssssss0 sssssssssNQQQQQ
```

Bcc_S

Function

16-Bit Branch

Extension Group

BASELINE

Operation

if (cc) PC = (PCL+s10)

Instruction Format

op s10

Syntax Example

BEQ_S s10

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

A branch is taken from the current PC with the displacement value specified in the source operand (rd) when one or more conditions are met, depending upon the instruction type used.

When using the B_S instruction, a branch is always executed from the current PC, 32-bit aligned, with the displacement value specified in the source operand.

For all branch types, the branch target is 16-bit aligned. The status flags are not updated with this instruction.

[Table 7-7](#) on page [308](#) lists the condition codes.



Caution

The Bcc_S instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

Pseudo Code

```

if cc==true then                                /* Bcc_S */
    KillDelaySlot(nPC)
    PC = cPCL + rd
else
    PC = nPC

```

Assembly Code Example

```

BEQ_S label    ; Branch to label if Z flag is set
BNE_S label    ; Branch to label if N flag is clear

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BEQ_S	s10	11110 01 ssssssss
BNE_S	s10	11110 10 ssssssss
BGT_S	s7	1111011 000 ssssss
BGE_S	s7	1111011 001 ssssss
BLT_S	s7	1111011 010 ssssss
BLE_S	s7	1111011 011 ssssss
BHI_S	s7	1111011 100 ssssss
BHS_S	s7	1111011 101 ssssss
BLO_S	s7	1111011 110 ssssss
BLS_S	s7	1111011 111 ssssss

BCLR

Function

Bit Clear

Extension Group

BASELINE

Operation

if (cc) then $a = (b \& \sim(1 \ll c));$

Instruction Format

op a, b, c

Syntax Example

BCLR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Clear (0) an individual bit within the value that is specified by the source operand 1 (b). The source operand 2 (c) contains a value that explicitly defines the bit-position that is to be cleared in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The result is written into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* BCLR */
  dest = src1 AND NOT(1 << (src2 & 31))
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]

```

Assembly Code Example

```
BCLR r1,r2,r3    ; Clear bit r3 of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BCLR<.f>	a,b,c	00100bbb00010000FBBBCCCCCAAAAA
BCLR<.f>	a,b,u6	00100bbb01010000FBBBuuuuuuAAAAA
BCLR<.f>	b,b,s12	00100bbb10010000FBBBssssssSSSSSS
BCLR<.cc><.f>	b,b,c	00100bbb11010000FBBBCCCCC0QQQQQ
BCLR<.cc><.f>	b,b,u6	00100bbb11010000FBBBuuuuuu1QQQQQ
BCLR_S	b,b,u5	10111bbb101uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BI

Function

Branch Indexed 32-bit Full-Word Table

Extension Group

CODE_DENSITY

Operation

$$PC = next\_PC + (c \ll 2)$$

Instruction Format

op c

Syntax Example

BI [r1]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The relative branch offset is defined by shifting the contents of the (c) operand register left by 2 places. Program execution resumes at the location given by the next sequential PC plus the branch offset.

Pseudo Code

```
PC = next_PC + (c << 2)          /* BI */
```

Assembly Code Example

```
BI [r1]      ;   branch to next_PC+r1<<2
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BI [c] 00100RRR001001000RRRCCCCCRRRRRR

BIC

Function

Bitwise AND Operation with Inverted Source

Extension Group

BASELINE

Operation

if (cc) $a = b \& \sim c$;

Instruction Format

op a, b, c

Syntax Example

BIC<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise AND of source operand 1 (b) with the inverse of source operand 2 (c) with the result written to the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                     /* BIC */
  dest = src1 AND NOT src2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
BIC r1,r2,r3    ; AND r2 with the NOT of r3 and write result into r1
```


Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BIC<.f>	a, b, c	00100bbb00000110FBBBCCCCCAAAAAA
BIC<.f>	a, b, u6	00100bbb01000110FBBBuuuuuuAAAAAA
BIC<.f>	b, b, s12	00100bbb10000110FBBBssssssSSSSSS
BIC<.cc><.f>	b, b, c	00100bbb11000110FBBBCCCCC0QQQQQ
BIC<.cc><.f>	b, b, u6	00100bbb11000110FBBBuuuuuu1QQQQQ
BIC_S	b, b, c	01111bbbccc00110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BIH

Function

Branch Indexed to Half-Word Table

Extension Group

CODE_DENSITY

Operation

$$PC = next\_PC + (c \ll 1)$$

Instruction Format

op c

Syntax Example

BIH [r1]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The relative branch offset is defined by shifting the contents of the (c) operand register to left by one place. Program execution resumes at the location given by the next sequential PC + the branch offset.

Pseudo Code

```
PC = next_PC + (c << 1)          /* BIH */
```

Assembly Code Example

```
BIH [r1]      ; branch to next_PC + r1 << 1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BIH [c] 00100RRR001001010RRRCCCCRRRRRR

BLcc

Function

Branch and Link

Extension Group

BASELINE

Operation

if (cc) {BLINK = NEXT_PC; PC = PCL +s21;}

Instruction Format

op s21

Syntax Example

BL<.cc><.d> s21

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

When a conditional branch and link is used and the specified condition is met (cc), program execution is resumed at the PCL (the PC, 32-bit aligned) plus a relative displacement, where PC is the address of the BLcc instruction. At the same time, the return address is stored in the link register BLINK (R31). This address is taken either from the first instruction following the branch (current PC) or the instruction after that (next PC) according to the delay slot mode (.d).

The delay slot modes, .d, are described in [Table 7-36](#) on page 332.

The condition codes, cc, are described in [Table 7-7](#) on page 308.



Caution

The BLcc and BL_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The conditional branch and link instruction has a maximum branch range of ± 1 M. The unconditional branch far format has a maximum branch range of ± 16 M. The target address for any branch and link instruction must be 32-bit aligned.

Because the execution of the instruction that is in the delay slot is controlled by the delay slot mode, the instruction must never be the target of any branch or jump instruction. The status flags are not updated with this instruction.



Note

The 16-bit-encoded instructions have a target address aligned to 32 bits. For this reason, a special encoding allows for a larger branch displacement. For example, BL_S s13 only needs to encode the top 11 bits because the bottom 2 bits of s13 are always zero because of the 32-bit data alignment.

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- ❑ Another jump or branch instruction
- ❑ Conditional loop instruction (LPcc)
- ❑ Return from interrupt ([RTIE](#))
- ❑ Any instruction with long-immediate data as a source operand

Pseudo Code

```
if cc==true then    /* BLcc */
  if N=1 then       /*nPC = instruction after BLcc
    BLINK = dPC      /*dPC= instruction after delay slot
    DelaySlot(nPC)   /*cPCL=BLcc instruction address, 32-bit aligned (PCL)
  else              /*N: delay slot mode (.d)
    BLINK = nPC
    PC = cPCL + rd
  else
    PC = nPC
```

Assembly Code Example

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

```
BLEQ label        ; if the Z flag is set then branch and link to label and
                   ;store the return address in BLINK
```

Syntax and Encoding

Instruction Code		
BL<.cc><.d>	s21	00001ssssssssss00SSSSSSSSSSNQQQQQ
BL<.d>	s25	00001ssssssssss10SSSSSSSSSSNRtttt
BL_S	s13	11111ssssssssssss

BMSK

Function

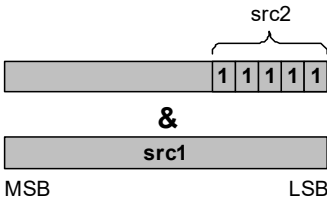
Bit Mask

Extension Group

BASELINE

Operation

if (cc) then a = (b & ((1 << (c+1))-1);



Instruction Format

op a, b, c

Syntax Example

BMSK<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Source operand 2 (c) specifies the size of a 32-bit mask value in terms of logical 1's starting from the LSB of a 32-bit register up to and including the bit specified by operand 2 (c). Only the bottom 5 bits of src2 are used as the bit value.

A logical AND is performed with the mask value and source operand (b). The result is written into the destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* BMSK */
    dest = src1 AND ((1 << ((src2 &
31)+1))-1)
    if F==1 then
        Z_flag = if dest==0 then 1 else
0
        N_flag = dest[31]

```

Assembly Code Example

```

BMSK r1,r2,31      ; Do not mask any bits of r2, write result into r1
BMSK r1,r2,7       ; Mask out the top 24 bits of r2 and write result into r1
BMSK r1,r2,0       ; Mask out all except bit 0 of r2 and write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BMSK<.f>	a,b,c	00100bbb00010011FBBBCCCCCAAAAAA
BMSK<.f>	a,b,u6	00100bbb01010011FBBBuuuuuuAAAAAA
BMSK<.f>	b,b,s12	00100bbh10010011FBBBssssssSSSSSS
BMSK<.cc><.f>	b,b,c	00100bbb11010011FBBBCCCCC0QQQQQ
BMSK<.cc><.f>	b,b,u6	00100bbb11010011FBBBuuuuuu1QQQQQ
BMSK_S	b,b,u5	10111bbh110uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BMSKN

Function

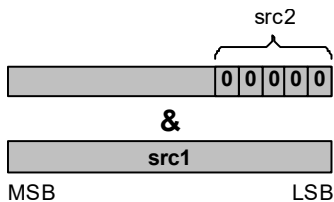
Bit Mask Negated

Extension Group

BASELINE

Operation

if (cc) then a = (b & (~(1 << (c+1))-1));



Instruction Format

op a, b, c

Syntax Example

BMSKN <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Source operand 2 (c) specifies the size of a 32-bit mask value in terms of logical 0's starting from the LSB of a 32-bit register up to and including the bit specified by operand 2(c). All bits beyond that position in the mask are 1's. Only the bottom 5 bits of src2 are used to define the size of the zero portion of the bit mask.

A logical AND is performed on the mask value and source operand (b). The result is written into the destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /*BMSKN */
  dest = src1 AND ~(1 << ((src2 & 31)+1))-1)
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]

```

Assembly Code Example

```
BMSKN r1,r2,8 ; r1 is set to (r2 & 0xFFFFFE0)
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BMSKN<.f>	a,b,c	00100bbb00101100FBBBCCCCC	AAAAAA
BMSKN<.f>	a,b,u6	00100bbb01101100FBBBuuuuuu	AAAAAA
BMSKN<.f>	b,b,s12	00100bbb10101100FBBBssssss	SSSSSS
BMSKN<.cc><.f>	b,b,c	00100bbb11101100FBBBCCCCC	0QQQQQ
BMSKN<.cc><.f>	b,b,u6	00100bbb11101100FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BRcc

Function

Compare and Branch

Extension Group

BASELINE for 32-bit BRcc operations

Operation

if (cc(b,c)) PC = PCL+s9

Instruction Format

op b, c, s9

Syntax Example

BREQ<.d><.T> b, c, s9

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

The BRcc instruction performs one of six possible 32-bit relational tests on its operands, and if the result is true, the branch is taken. If a delay-slot is specified, using delay slot mode shown in [Table 7-36](#) on page 332, one instruction immediately following the branch in program order is executed before the branch takes effect.

The branch target is computed as the sum of the branch’s 32-bit word-aligned PC plus the 16-bit half-word aligned displacement value given by the signed 9-bit literal operand (rd).

The available branch conditions are shown in the following table:

Instruction	Description	Branch Condition
BREQ	Branch if Equal	if (b==c) PC = (PCL +s9)
BRNE	Branch if Not Equal	if (b!=c) PC = (PCL + s9)
BRLT	Branch if Less Than (Signed)	if (b<c) PC = (PCL + s9)

BRGE	Branch if Greater Than or Equal (Signed)	if (b>=c) PC = (PCL + s9)
BRLO	Branch if Lower Than (Unsigned)	if (b<c) PC = (PCL + s9)
BRHS	Branch if Higher Than or Same (Unsigned)	if (b>=c) PC = (PCL + s9)

For 16-bit compare and branch instructions, BRNE_S compares the source operand 1 (b) with 0, and if src1 is not equal to zero, the branch is taken.

BREQ_S performs the same comparison. However, the branch is taken when the source operand 1 (b) is equal to zero.

Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction.

The status flags are not updated by this instruction.



Caution

The BRcc and BRcc_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

Some implementations of the ARCV2 may predict the outcome of BRcc instructions. These predictions may be static, that is, based on the instruction pattern, whereas others may be dynamic, that is, based on the observed execution history of the branch. A description of the static prediction semantics for BRcc instructions can be found in section “[Static Branch Prediction Mode](#)”, and more details can be found in section “[Branch On Compare or Bit Test](#)”. The interpretation of the <T> syntax can be found in [Table 7-9](#) on page 310.

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code

BREQ <.d><.T>	b, c, s9	00001bbbssssss1SBBBCCCCCN0Y000
BREQ <.d><.T>	b, u6, s9	00001bbbssssss1SBBBuuuuuuN1Y000
BRNE <.d><.T>	b, c, s9	00001bbbssssss1SBBBCCCCCN0Y001
BRNE <.d><.T>	b, u6, s9	00001bbbssssss1SBBBuuuuuuN1Y001
BRLT <.d><.T>	b, c, s9	00001bbbssssss1SBBBCCCCCN0Y010
BRLT <.d><.T>	b, u6, s9	00001bbbssssss1SBBBuuuuuuN1Y010
BRGE <.d><.T>	b, c, s9	00001bbbssssss1SBBBCCCCCN0Y011
BRGE <.d><.T>	b, u6, s9	00001bbbssssss1SBBBuuuuuuN1Y011

BRLO <.d><.T>	b, c, s9	00001bbbssssssss1SBBBCCCCCN0Y100
BRLO <.d><.T>	b, u6, s9	00001bbbssssssss1SBBBuuuuuuN1Y100
BRHS <.d><.T>	b, c, s9	00001bbbssssssss1SBBBCCCCCN0Y101
BRHS <.d><.T>	b, u6, s9	00001bbbssssssss1SBBBuuuuuuN1Y101
BRNE_S	b, 0, s8	11101bbb1ssssssss
BREQ_S	b, 0, s8	11101bbb0ssssssss

Additional Pseudo Syntax and Encoding

The following pseudo-instructions for missing conditions are available using existing encodings:

BRGT<.d> b, c, s9	Encode as BRLT<.d> c, b, s9
BRGT<.d> b, u6, s9	Encode as BRGE<.d> b, u6+1, s9
BRHI<.d> b, c, s9	Encode as BRLO<.d> c, b, s9
BRHI<.d> b, u6, s9	Encode as BRHS<.d> b, u6+1, s9
BRLE<.d> b, c, s9	Encode as BRGE<.d> c, b, s9
BRLE<.d> b, u6, s9	Encode as BRLT<.d> b, u6+1, s9
BRLS<.d> b, c, s9	Encode as BRHS<.d> c, b, s9
BRLS<.d> b, u6, s9	Encode as BRLO b, u6+1, s9

BRK

Function

Breakpoint

Extension Group

BASELINE

Operation

Halt and flush the processor

Instruction Format

op

Syntax Example

BRK_S

Flag Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged
BH	☒	= 1
H	☒	= 1

Description

The breakpoint instruction halts the processor without advancing the program counter or otherwise modifying the state of the processor or its memory.

The BRK_S instruction is normally used by an external debug host to set breakpoints during debugging. The debugger typically replaces the instruction at the breakpoint address with the BRK_S instruction to force the processor to halt whenever PC reaches the breakpoint address. To restart the processor after taking a breakpoint, the debugger must write the original instruction back to memory and issue an instruction cache invalidation command. The external debug host can then restart the processor, and resume normal execution of the original instruction from the breakpoint address.

There is no limit to the number of breakpoints that can be inserted into a piece of code.



The breakpoint instruction sets the BH bit in the Debug register, which allows the debugger to determine what caused the ARCV2-based processor to halt. The BH bit is cleared when the Halt bit in the Status register is cleared, for example, by restarting or single-stepping the ARCV2-based processor.

The breakpoint instruction is a kernel-only instruction unless enabled by the UB bit in the DEBUG register. Any attempt to execute BRK_S, with the UB bit set to 0, raises a privilege violation exception (see [Machine Check, Internal Instruction Memory Error](#)).

The ARCV2 provides both a 32-bit (BRK) and a 16-bit (BRK_S) encoding of the breakpoint instruction.

The breakpoint instruction can be placed anywhere in a program, including the delay slot of branch or jump instructions, and also immediately following a BRcc, a BBIT0, or a BBIT1 instruction.

Pseudo Code

```
FlushPipe()                      /* BRK_S, BRK */
DEBUG[BH] = 1
Halt()
```

Assembly Code Example

A breakpoint instruction may be inserted into any position.

```
MOV    r0, 0x04
ADD    r1, r0, r0
XOR.F  0, r1, 0x8
BRK_S                                     ; ←--- break here
SUB    r2, r0, 0x3
ADD.NZr1, r0, r0
JZ.D   [r8]
OR     r5, r4, 0x10
```

Breakpoints may be inserted at a jump delay slot.

```
MOV    r0, 0x04
ADD    r1, r0, r0
XOR.F  0, r1, 0x8
SUB    r2, r0, 0x3
ADD.NZr1, r0, r0
JZ.D   [r8]
BRK_S                                     ; ←--- break here
RLC    r2, r1
```


Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BRK	0010010101101111000RRRRRR111111
BRK_S	0111111111111111



Note

By software convention, the reserved fields. *b* and *c* are set to 0x7 for the BRK_S instruction (the encoding is 0x7FFF).

BSET

Function

Bit Set

Extension Group

BASELINE

Operation

if (cc) a = (b | (1<<c));

Instruction Format

op a, b, c

Syntax Example

BSET<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Set (to 1) an individual bit within the value that is specified by source operand 1 (b). Source operand 2 (c) contains a value that explicitly defines the bit-position that is to be set in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The result is written into the destination register (dest). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* BSET */
  dest = src1 OR (1 << (src2 & 31))
  if F==1 then
    Z_flag = 0
    N_flag = dest[31]

```

Assembly Code Example

```
BSET r1,r2,r3    ; Set bit r3 of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BSET<.f>	a,b,c	00100bbb00001111FBBBCCCCCAAAAA
BSET<.f>	a,b,u6	00100bbb01001111FBBBuuuuuuAAAAA
BSET<.f>	b,b,s12	00100bbb10001111FBBBssssssSSSSSS
BSET<.cc><.f>	b,b,c	00100bbb11001111FBBBCCCCC0QQQQQ
BSET<.cc><.f>	b,b,u6	00100bbb11001111FBBBuuuuuu1QQQQQ
BSET_S	b,b,u5	10111bbb100uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BSPEEK

Function

Peek the head of the bitstream

Extension Group

Bit-buffer, `has_bs==true`

Operation

`b=peek(c) ;`

Instruction Format

`op b, c`



Note

You cannot use the XY operands as a source or destination with this instruction.

Syntax Example

`BSPEEK<.F> b,c`

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

This instruction return *N* bits from the head of the bitstream buffer in the XY memory (or DCCM). *N* is defined in the input argument *C*.

This instruction prefetches data from the bitstream buffer. At the beginning of a new stream or on a context switch the prefetched data should be invalidated by setting the `BS_AUX_CTRL.IV` bit.

When used with the `.F` suffix, the instruction sets the Z flag if the result is zero else it clears the Z flag.

Pseudo Code

```

// get base address                                     /*BSPEEK
base_addr  = BS_AUX_ADDR
// get value of bit offset
bit_offset = BS_AUX_BIT_OS;
// bit offset within word
bit_os     = bit_offset & 0x1f;
// get size of buffer
buf_mask   = (1 << (BS_AUX_CTRL.SIZE+10)) - 1;
// 32b aligned byte address
mem_addr   = &BS_AUX_ADDR[(bit_offset>>5)<<2];
// next 32b aligned address
mem_addr_nxt = ((mem_addr + 4) & buf_mask) | (base_addr & ~buf_mask);
// c unsigned
nbits = c < 32 ? c : 32;
v = 0;
if (BS_AUX_CTRL.BO) {
    // msb is at head of stream
    v = (v << 8) | mem_addr[0];
    v = (v << 8) | mem_addr[1];
    v = (v << 8) | mem_addr[2];
    v = (v << 8) | mem_addr[3];
    v = (v << 8) | mem_addr_nxt[0];
    v = (v << 8) | mem_addr_nxt[1];
    v = (v << 8) | mem_addr_nxt[2];
    v = (v << 8) | mem_addr_nxt[3];
    v >>= (64 - nbits - bit_os);
} else {
    // lsb is at head of stream
    v = (v << 8) | mem_addr_nxt[3];
    v = (v << 8) | mem_addr_nxt[2];
    v = (v << 8) | mem_addr_nxt[1];
    v = (v << 8) | mem_addr_nxt[0];
    v = (v << 8) | mem_addr[3];
    v = (v << 8) | mem_addr[2];
    v = (v << 8) | mem_addr[1];
    v = (v << 8) | mem_addr[0];
    v >>= bit_os;
}
// mask bits
b &= (1 << nbits) - 1;
if (F == 1) {
    Z_flag = (b == 0);
}

```

Assembly Code Example

```
BSPEEK r0,r1      ; Return N bits from head of Bitstream in r0; N from r1
```

Syntax and Encoding

Instruction Code		
BSPEEK<.F>	b, c	00101bbb00101111FBBBCCCCC101110
BSPEEK<.F>	b, u6	00101bbb01101111FBBBuuuuuu101110

BSPOP

Function

Pop head of the bitstream

Extension Group

Bit-buffer, has_bs==true

Operation

```
b=peek(c);
BS_AUX_BIT_OS = wrap(BS_AUX_BIT_OS, c, BS_AUX_CTRL.SIZE);
```

Instruction Format

op b, c

Syntax Example

BSPOP<.F> b,c

You cannot use the XY operands as a source or destination with this instruction.

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

This instruction return N bits from the head of the bitstream buffer in the XY memory (or DCCM). N is defined in the input argument C .

This instruction prefetches data from the bitstream buffer. At the beginning of a new stream or on a context switch the prefetched data should be invalidated by setting the `BS_AUX_CTRL.IV` bit.

When used with the `.F` suffix, the instruction sets the Z flag if the result is zero else it clears the Z flag.

Pseudo Code

```

// get base address                                     /* BSPOP */
base_addr  = BS_AUX_ADDR
// get value of bit offset
bit_offset = BS_AUX_BIT_OS;
// bit offset within word
bit_os     = bit_offset & 0x1f;
// get size of buffer
buf_mask   = (1 << (BS_AUX_CTRL.SIZE+10)) - 1;
// 32b aligned byte address
mem_addr   = &BS_AUX_ADDR[(bit_offset>>5)<<2];
// next 32b aligned address
mem_addr_nxt = ((mem_addr + 4) & buf_mask) | (base_addr & ~buf_mask);
// c unsigned
nbits = c < 32 ? c : 32;
v = 0;
if (BS_AUX_CTRL.BO) {
    // msb is at head of stream
    v = (v << 8) | mem_addr[0];
    v = (v << 8) | mem_addr[1];
    v = (v << 8) | mem_addr[2];
    v = (v << 8) | mem_addr[3];
    v = (v << 8) | mem_addr_nxt[0];
    v = (v << 8) | mem_addr_nxt[1];
    v = (v << 8) | mem_addr_nxt[2];
    v = (v << 8) | mem_addr_nxt[3];
    v >>= (64 - nbits - bit_os);
} else {
    // lsb is at head of stream
    v = (v << 8) | mem_addr_nxt[3];
    v = (v << 8) | mem_addr_nxt[2];
    v = (v << 8) | mem_addr_nxt[1];
    v = (v << 8) | mem_addr_nxt[0];
    v = (v << 8) | mem_addr[3];
    v = (v << 8) | mem_addr[2];
    v = (v << 8) | mem_addr[1];
    v = (v << 8) | mem_addr[0];
    v >>= bit_os;
}
// mask bits
b &= (1 << nbits) - 1;
// update bit offset
bit_offset += nbits
bit_offset &= (buf_mask << 3) | 7;
BS_AUX_BIT_OS = bit_offset
if (F == 1) {
    Z_flag = (b == 0);
}

```


Assembly Code Example

```
BSPOP r0,r1      ; Return N bits from head of Bitstream in r0; N from r1
                  ; Update the pointer to the head of the buffer.
```

Syntax and Encoding

Instruction Code

BSPOP<.F>	b,c	00101 bbb 00 101111 F BBB CCCCC 101111
BSPOP<.F>	b,u6	00101 bbb 01 101111 F BBB uuuuuu 101111

BSPUSH

Function

Push to the tail of the bitstream

Extension Group

Bit-buffer, has_bs==true

Operation

```
push(b, c);
BS_AUX_BIT_OS = wrap(BS_AUX_BIT_OS, c, BS_AUX_CTRL.SIZE);
```

Instruction Format

op a, b, c

Syntax Example

BSPUSH a,b,c

You cannot use the XY operands as a source or destination with this instruction.

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Push N bits of value B to the tail of the bitstream buffer in the XY memory (or DCCM). N is defined in the input argument C . After the push, the pointer is updated to the head of the bitstream buffer. The new bit offset value is stored in the BS_AUX_OS register. The head of the buffer is cached in the BS_AUX_WDATA register. The end of the stream needs to insert some dummy bits to flush the contents of the register to memory.

Pseudo Code

```

// get base address
base_addr = BS_AUX_ADDR
// get value of bit offset
bit_offset = BS_AUX_BIT_OS;
// bit offset within word
bit_os = bit_offset & 0x1f;
// get size of buffer
buf_mask = (1 << (BS_AUX_CTRL.SIZE+10)) - 1;
// 32b aligned byte address
mem_addr = &BS_AUX_ADDR[(bit_offset>>5)<<2];
// next 32b aligned address
mem_addr_nxt = ((mem_addr + 4) & buf_mask) | (base_addr & ~buf_mask);
// c unsigned
nbits = c < 32 ? c : 32;
// mask for new bits
m = (1 << nbits) - 1;
val64 = m & b;
if (BS_AUX_CTRL.BO) {
    // msb is at head of stream
    v = BS_AUX_WDATA << 32;
    // replace bits at bottom of v
    m = m << (64 - nbits - bit_os);
    v = (v & ~m) | (val64 << (64 - nbits - bit_os));
    bit_os += nbits
    if (bit_os >= 32) {
        // write-back to memory
        BS_AUX_WDATA = v;
        v >>= 32
        for (i = 0; i < 4; i++) {
            mem_addr[3-i] = v;
            v >>= 8;
        }
    } else {
        BS_AUX_WDATA = v >> 32;
    }
} else {
    // lsb is at head of stream
    v = BS_AUX_WDATA
    // replace bits at top of v
    m = m << bit_os;
    v = (v & ~m) | (val64 << bit_os);
    bit_os += nbits
    if (bit_os >= 32) {
        // write-back to memory
        for (i = 0; i < 4; i++) {
            mem_addr[3-i] = v;
            v >>= 8;
        }
    }
    BS_AUX_WDATA = v;
}
bit_offset += nbits
bit_offset &= (buf_mask << 3) | 7;
BS_AUX_BIT_OS = bit_offset;
a = bit_offset

```

Assembly Code Example

```
BSPUSH r0,r1      ; Push r1 bits from r0 at tail of Bitstream
```

Syntax and Encoding

Instruction Code		
BSPUSH	a, b, c	00101bbb001011101BBBCCCCCAAAAAA
BSPUSH	a, b, u6	00101bbb011011101BBBuuuuuuAAAAAA
BSPUSH	b, b, s12	00101bbb101011101BBBssssssSSSSSS
BSPUSH<.cc>	b, b, c	00101bbb111011101BBBCCCCC0QQQQQ
BSPUSH<.cc>	b, b, u6	00101bbb111011101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BTST

Function

Bit Test

Extension Group

BASELINE

Operation

if (cc) (b & (1 << c))

Instruction Format

op b, c

Syntax Example

BTST<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logically AND source operand 1 (b) with a bit mask specified by source operand 2 (c). Source operand 2 (c) explicitly defines the bit that is tested in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The flags are updated to reflect the result. The flag setting field, F, is always encoded as 1 for this instruction.

There is no result write-back.



Note

BTST and BTST_S always set the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                                /* BTST */
    alu = src1 AND (1 << (src2 & 31))
    Z_flag = if alu==0 then 1 else 0
    N_flag = alu[31]
```

Assembly Code Example

```
BTST r2,28    ; Test bit 28 of r2 and update flags on result
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BTST	b, c	00100bbb000100011BBBCCCCCRRRRRR
BTST	b, u6	00100bbb010100011BBBuuuuuuRRRRRR
BTST	b, s12	00100bbb100100011BBBssssssSSSSSS
BTST<.cc>	b, c	00100bbb110100011BBBCCCCC0QQQQQ
BTST<.cc>	b, u6	00100bbb110100011BBBuuuuuu1QQQQQ
BTST_S	b, u5	10111bbb111uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BXOR

Function

Bit Exclusive OR (Bit Toggle)

Extension Group

BASELINE

Operation

if (cc) $a = (b \wedge (1 \ll c));$

Instruction Format

op a, b, c

Syntax Example

BXOR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logically XOR source operand 1 (b) with a bit mask specified by source operand 2 (c). Source operand 2 (c) explicitly defines the bit that is to be toggled in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The result is written to the destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* BXOR */
  dest = src1 XOR (1 << (src2 & 31))
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]

```


Assembly Code Example

```
BXOR r1,r2,r3      ; Toggle bit r3 of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BXOR<.f>	a,b,c	00100bbb00010010FBBBCCCCC	AAAAAA
BXOR<.f>	a,b,u6	00100bbb01010010FBBBuuuuuu	AAAAAA
BXOR<.f>	b,b,s12	00100bbb10010010FBBBssssss	SSSSSS
BXOR<.cc><.f>	b,b,c	00100bbb11010010FBBBCCCCC	0QQQQQ
BXOR<.cc><.f>	b,b,u6	00100bbb11010010FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

CLRI

Function

CLRI forces the STATUS32.IE bit to 0, disabling interrupts.

Extension Group

HAS_INTERRUPTS == 1

Operation

$\{\text{dest} \leftarrow \{ 26'd0, 1'b1, \text{STATUS32.IE}, \text{STATUS32.E}[3:0] \}; \text{STATUS32.IE} \leftarrow 0\}$

Instruction Format

op c

Syntax Example

CLRI c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

CLRI forces the STATUS32.IE bit to 0, disabling interrupts. If a destination register is specified as a c operand, this instruction saves the interrupt state from the STATUS32 register to the destination register before disabling interrupts.

The CLRI instruction is available only in kernel mode. Using this instruction in the user mode raises a [Privilege Violation, Kernel/User Only Access](#) exception. CLRI is a serializing instruction similar to the KFLAG instruction. This behavior means after this instruction commits, the pipeline is flushed.

If your processor includes the SecureShield component,

- ❑ the CLRI instruction is available only in the secure kernel mode if SEC_STAT.NIC==0. Usage of this instruction in any other mode raises a Privilege Violation exception (0x071000).
- ❑ the CLRI instruction is available in the secure kernel and normal kernel modes if SEC_STAT.NIC==1.
- ❑ Using the CLRI instruction in the secure or normal user mode raises a Privilege Violation exception (0x071020).

Pseudo Code

```

{                                                                 /* CLRI */
    dst <- { 26'd0, 1'b1, STATUS32.IE, STATUS32.E[3:0] }
    STATUS32.IE <- 0
}

if (SEC_MODES_OPTION == 1) /* SecureShield is present */
if ((STATUS32.U == 0) && ((STATUS32.S == 1) || (SEC_STAT.NIC == 1))) /* CLRI * if
dst = (0 | (1 << 6) | (STATUS32.IE << 5) | (STATUS32.E))           SecureShield
STATUS32.IE = 0                                                    component is
else /* SecureShield is not present */                             configured.
if (STATUS32.U == 0) /* in kernel mode */
    dst = (0 | (1 << 6) | (STATUS32.IE << 5) | (STATUS32.E))
    STATUS32.IE = 0

```

Assembly Code Example

```
CLRI R0      ;clear interrupts and capture interrupt state in R0
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

CLRI	c	00100111001011110000cccccc111111
CLRI	u6	00100111011011110000uuuuuu111111

If the destination register value is 62, or if a u6 operand is specified, CLRI does not save the interrupt state to the destination register value.



Note

Use of limm operands with the CLRI instruction is deprecated. If you use a limm operand with the CLRI instruction, an Illegal Instruction exception is raised.

If a destination C register, whose encoded value is not 62, is specified, CLRI saves the interrupt state from the STATUS32 register to the destination register before disabling interrupts. The resulting destination register can be subsequently used as an operand to the SETI instruction to restore the interrupt enable and interrupt priority level to their values prior to execution of the CLRI instruction. The 32-bit destination register is assigned the following values:

dst [31:6]	dst [5]	dst[4]	dst[3:0]
26'd0	1	STATUS32.IE	STATUS32.E[3:0]

CMP

Function

Comparison

Extension Group

BASELINE

Operation

if (cc) $b - c$;

Instruction Format

op b, c

Syntax Example

CMP<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

A comparison is performed by subtracting source operand 2 (c) from source operand 1 (b) and subsequently updating the flags.



Note

The CMP instruction always sets the flags even though there is no associated flag setting suffix.

There is no destination register. Therefore, the result of the subtraction is discarded.

Pseudo Code

```

if cc==true then                                /* CMP */
    alu = src1 - src2
    Z_flag = if alu==0 then 1 else 0
    N_flag = alu[31]
    C_flag = Carry()
    V_flag = Overflow()

```

Assembly Code Example

```

CMP r1,r2    ; Subtract r2 from r1 and set the flags on the result

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

CMP and CMP_S always set the flags even though there is no associated flag setting suffix.

Instruction Code

CMP	b,c	00100bbb000011001BBBCCCCCRRRRRR
CMP	b,u6	00100bbb010011001BBBuuuuuuRRRRRR
CMP	b,s12	00100bbb100011001BBBssssssSSSSSS
CMP<.cc>	b,c	00100bbb110011001BBBCCCCC0QQQQQ
CMP<.cc>	b,u6	00100bbb110011001BBBuuuuuu1QQQQQ
CMP_S	b,h	01110bbbhhh100HH
CMP_S	h,s3	01110ssshhh101HH
CMP_S	b,u7	11100bbb1uuuuuuu

DADDH11

Function

Add to D1, writing to D1

Extension Group

Double-precision Floating Point

Operation

$$D1 = D1 + (src1:src2)$$

$$dest = (D1 + (src1:src2)).high$$

Instruction Format

```
inst dest, src1, src2
```

Format Key:

dest = Destination Register

src2 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point add of data register D1 to the 64-bit value formed from operand 1 and operand 2. The result is stored in data register D1 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D1 = D1 + (src1:src2)          /*DADDH11*/
dest = (D1 + (src1:src2)).high
```

Assembly Code Example

```
DADDH11 r1,r2,r3      ;Double-precision floating point add of data register D1
                      ;to the 64-bit value formed from operand 1 and operand 2.
```

Syntax and Encoding

Instruction Code

With Result

DADDH11<.f>	a,b,c	00110 bbb 00 110100 F BBB CCCCC AAAAAA
DADDH11 <.f>	a,b,limm	00110 bbb 00 110100 F BBB 111110 AAAAAA
DADDH11 <.f>	a,limm,c	00110 110 00 110100 F 111 CCCCC AAAAAA

Without Result

DADDH11<.f>	0,b,c	00110 bbb 00 110100 F BBB CCCCC 111110
DADDH11 <.f>	0,b,limm	00110 bbb 00 110100 F BBB 111110 111110
DADDH11 <.f>	0,limm,c	00110 110 00 110100 F 111 CCCCC 111110

Related Instructions

[DADDH12](#) [DADDH22](#)

[DADDH21](#) [DSUBH11](#)

DADDH12

Function

Add to D2, writing to D1

Extension Group

Double-precision Floating Point

Operation

$$D1 = D2 + (src1:src2)$$

$$dest = (D2 + (src1:src2)).high$$

Instruction Format

```
inst dest, src1, src2
```

Format Key:

dest = Destination Register

src2 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero.	L	= Limm Data
N	•	= Set if MSB of write-back result is set.		
C	•	= Set if write-back result is a NaN.		
V	•	= Set if write-back result overflowed.		

Description:

Double-precision floating point add of data register D2 to the 64-bit value formed from operand 1 and operand 2. The result is stored in data register D1 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D1 = D2 + (src1:src2)          /*DADDH12*/
dest = (D2 + (src1:src2)).high
```

Assembly Code Example

```
DADDH12 r1,r2,r3      ;Double-precision floating point add of data
                      ;register D2 to the 64-bit value formed from
                      ;operand 1 and operand 2.
```

Syntax and Encoding

Instruction Code

With Result

DADDH12<.f>	a,b,c	00110 bbb 00 110101 F BBB CCCCC AAAAAA
DADDH12 <.f>	a,b,limm	00110 bbb 00 110101 F BBB 111110 AAAAAA
DADDH12 <.f>	a,limm,c	00110 110 00 110101 F 111 CCCCC AAAAAA

Without Result

DADDH12 <.f>	0,b,c	00110 bbb 00 110101 F BBB CCCCC 111110
DADDH12 <.f>	0,b,limm	00110 bbb 00 110101 F BBB 111110 111110
DADDH12 <.f>	0,limm,c	00110 110 00 110101 F 111 CCCCC 111110

Related Instructions

[DBNZ](#) [DMULH12](#)
[DADDH21](#) [DSUBH12](#)
[DADDH22](#)

DADDH21

Function

Add to D1, writing to D2

Extension Group

Double-precision Floating Point

Operation

$$D2 = D1 + (src1:src2)$$

$$dest = (D1 + (src1:src2)).high$$

Instruction Format

```
inst dest, src1, src2
```

Format Key:

dest = Destination Register

src1 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point add of data register D1 to the 64-bit value formed from operand 1 and operand 2. The result is stored in data register D2 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D2 = D1 + (src1:src2) /*DADDH21*/
dest = (D1 + (src1:src2)).high
```

Assembly Code Example

```
DADDH21 r1,r2,r3 ;Double-precision floating point add of data register D1
                 ;to the 64-bit value formed from operand 1 and operand 2.
```

Syntax and Encoding

Instruction Code

With Result

DADDH21<.f>	a,b,c	00110bbb00110110FBBBCCCCCAAAAAA
DADDH21 <.f>	a,b,limm	00110bbb00110110FBBB111110AAAAAA
DADDH21 <.f>	a,limm,c	0011011000110110F111CCCCCAAAAAA

Without Result

DADDH21 <.f>	0,b,c	00110bbb00110110FBBBCCCCC111110
DADDH21 <.f>	0,b,limm	00110bbb00110110FBBB111110111110
DADDH21 <.f>	0,limm,c	0011011000110110F111CCCCC111110

Related Instructions

[DBNZ](#) [DMULH21](#)

[DADDH12](#) [DSUBH21](#)

[DADDH22](#)

DADDH22

Add to D2, writing to D2

Extension Group

Double-precision Floating Point

Operation

```
D2 = D2 + (src1:src2)
dest = (D2 + (src1:src2)).high
```

Instruction Format

```
inst dest, src1, src2
```

Format Key:

```
dest    = Destination Register
src1     = Source Operand 1
src2     = Source Operand 2
```

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description:

Double-precision floating point add of data register D2 to the 64-bit value formed from operand 1 and operand 2. The result is stored in data register D2 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D2 = D2 + (src1:src2) /*DADDH22*/
dest = (D2 + (src1:src2)).high
```

Assembly Code Example

```
DADDH22 r1,r2,r3 ;Double-precision floating point add of data register D2
                 ;to the 64-bit value formed from operand 1 and operand 2.
```

Syntax and Encoding

Instruction Code

With Result

DADDH22<.f>	a,b,c	00110 bbb 00 110111 F BBB CCCCC AAAAA
DADDH22 <.f>	a,b,limm	00110 bbb 00 110111 F BBB 111110 AAAAA
DADDH22 <.f>	a,limm,c	00110 110 00 110111 F 111 CCCCC AAAAA

Without Result

DADDH22 <.f>	0,b,c	00110 bbb 00 110111 F BBB CCCCC 111110
DADDH22 <.f>	0,b,limm	00110 bbb 00 110111 F BBB 111110 111110
DADDH22 <.f>	0,limm,c	00110 110 00 110111 F 111 CCCCC 111110

Related Instructions

[DBNZ](#) [DMULH22](#)

[DADDH12](#) [DSUBH22](#)

[DADDH21](#)

DEXCL1

Function

Register D1 Exchange

Extension Group

Double-precision Floating Point

Operation

```
dest = (D1).low
D1 = (src1:src2)
```

Instruction Format

```
inst dest, src1, src2
```

Format Key

- dest = Destination Register
- src1 = Source Operand 1
- src2 = Source Operand 2

Flag Affected (32-Bit)

Z

N

C

V

= Unaffected

= Unaffected

= Unaffected

= Unaffected

L

= Limm Data

Description

First, the lower half of data register D1 is copied to destination register. Then, the 64-bit value formed by operand 1 and operand 2 is copied to data register D1.

Pseudo Code

```
dest = (D1).low                                /*DEXCL1*/
D1 = (src1:src2)
```

Assembly Code Example

```
DEXCL1 r1,r2,r3    ;Register D1 Exchange
```

Syntax and Encoding

Instruction Code

With Result

DEXCL1	a,b,c	00110bbb001111000BBBCCCCCAAAAAA
DEXCL1	a,b,limm	00110bbb001111000BBB111110AAAAAA
DEXCL1	a,limm,c	00110110001111000111CCCCCAAAAAA

Without Result

DEXCL1	0,b,c	00110bbb001111000BBBCCCCC111110
DEXCL1	0,b,limm	00110bbb001111000BBB111110111110
DEXCL1	0,limm,c	00110110001111000111CCCCC111110

Related Instructions

[DEXCL2](#)

DEXCL2

Function

Register D2 Exchange

Extension Group

Double-precision Floating Point

Operation

```
dest = (D2).low
D2 = (src1:src2)
```

Instruction Format

```
inst dest, src1, src2
```

Format Key:

```
dest    = Destination Register
src1    = Source Operand 1
src2    = Source Operand 2
```

Flag Affected (32-Bit)

Z		= Unaffected	L	= Limm Data
N		= Unaffected		
C		= Unaffected		
V		= Unaffected		

Description

First, the lower half of data register D2 is copied to destination register. Then, the 64-bit value formed by operand 1 and operand 2 is copied to data register D2.

Pseudo Code

```
dest = (D2).low
D2 = (src1:src2)                                /*DEXCL2*/
```


Assembly Code Example

```
DEXCL2 r1,r2,r3    ;Register D2 Exchange
```

Syntax and Encodings

Instruction Code		
With Result		
DEXCL2	a,b,c	00110bbb001111010BBBCCCCCAAAAAA
DEXCL2	a,b,limm	00110bbb001111010BBB111110AAAAAA
DEXCL2	a,limm,c	00110110001111010111CCCCCAAAAAA
Without Result		
DEXCL2	0,b,c	00110bbb001111010BBBCCCCC111110
DEXCL2	0,b,limm	00110bbb001111010BBB111110111110
DEXCL2	0,limm,c	00110110001111010111CCCCC111110

Related Instructions

[DEXCL1](#)

DBNZ

Function

Decrement and Branch if Non-zero

Extension Group

BASELINE

Operation

if (src != 1) PC = PCL + s13; src = src - 1

Instruction Format

op b, s13

Syntax Example

DBNZ<.d> b, s13

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The DBNZ instruction decrements its source register operand (src), and if the result is non-zero it branches to the location defined by a signed half-word displacement operand (s13).

This instruction supports only the REG-S12 operand format, as it always requires one register and a signed 12-bit literal operand. Any use of other operand formats raises an Illegal Instruction exception. The register operand must be a writable general-purpose register, that is, it cannot be a long-immediate indicator (r62), nor can it be the PCL register (r63). Any attempt to use one of those register operands also raises an Illegal Instruction exception.

The 13-bit literal operand (s13) is encoded in the DBNZ instruction format as an s12 field in which the least-significant bit of the displacement has been omitted. This is because the displacement is always half-word aligned. The 13-bit relative displacement is added to the current 32-bit word-aligned PC value (PCL) to determine the branch target address.

When DBNZ is used to implement a loop-closing branch the target address represents the address of the first instruction of loop body.

The DBNZ instruction may optionally specify that a delay-slot instruction follows. The delay slot modes, .d, are described in [Table 7-36](#) on page 332.



Caution

The DBNZ instruction cannot appear in the delay slot of another branch or jump instruction, nor can it appear in the execution slot of an EI_S instruction.

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```

if (src != 1) {                                     /* DBNZ */
    if (Instruction[16])
        STATUS32.DE = 1;
        BTA = cPCL + offset
    } else {
        PC = cPCL + offset
    }
}
src = src - 1

```

Assembly Code Example

```

DBNZ r1, label          ; decrement r1, and branch to label if result
                        ; is non-zero

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```

DBNZ<.d>    b, s13    00100bbb1000110N0BBBssssssSSSSSS

```

DIV

Function

2's Complement Integer Divide. For the DSP version, see [DIV](#).

Extension Group

DIV_REM_OPTION!=none or DSP_DIVSQRT!=none

Operation

if (cc) then $a = b / c$;

Instruction Format

op a, b, c

Syntax Example

DIV <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if divisor is zero or if an arithmetic overflow occurs

Description

If the divisor (c) is non-zero, the destination register is assigned the quotient obtained by signed integer division of source operand (b) by source operand (c).

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an `EV_DivZero` exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

An arithmetic overflow is signaled if the resulting value cannot be represented in 32 bits. This overflow occurs only in one specific case, which is the division of the largest representable negative value (0x80000000) by -1. If an arithmetic overflow occurs, the overflow flag (V) is set to 1.

If a division-by-zero is attempted, or if the result cannot be represented in 32 bits, when the `EV_DivZero` exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, "Implementation Behavior on Division Overflow"](#).

Pseudo Code

```

if (cc == true) {
    if ((src2 != 0) && ((src1 != 0x80000000) || src2 != 0xffffffff))
    {
        dest = src1 / src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = dest[31];
            V = 0;
        }
    } else {
        if ((src2 == 0) && (STATUS32.DZ == 1))
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1) {
                V = 1;
            }
        }
    }
}

```

Assembly Code Example

```
DIV r1,r2,r3 ; r1 is assigned r2 / r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
DIV<.f>	a,b,c	00101bbb00000100FBBBCCCCC0AAAAAA
DIV<.f>	a,b,u6	00101bbb01000100FBBBuuuuuuAAAAAA
DIV<.f>	b,b,s12	00101bbb10000100FBBBssssssSSSSSS
DIV<.cc><.f>	b,b,c	00101bbb11000100FBBBCCCCC0QQQQQQ
DIV<.cc><.f>	b,b,u6	00101bbb11000100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DIVU

Function

Unsigned Integer Divide. For the DSP version, see [DIVU](#).

Extension Group

DIV_REM_OPTION!=none or DSP_DIVSQRT!=none

Operation

if (cc) then $a = b / c$;

Instruction Format

op a, b, c

Syntax Example

DIVU <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always cleared
C		= Unchanged
V	•	= Set if divisor is zero

Description

If the divisor (c) is non-zero, the destination register is assigned the quotient obtained by unsigned integer division of source operand (b) by source operand (c).

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an `EV_DivZero` exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

If a division-by-zero is attempted, when the `EV_DivZero` exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, "Implementation Behavior on Division Overflow"](#).

If the flag-update bit is set, an unsigned DIVU operation always clears the N-flag.

Pseudo Code

```

if (cc == true) {
    if (src2 != 0) {
        dest = src1 / src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = 0;
            V = 0;
        }
    } else {
        if (STATUS32.DZ == 1)
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1) {
                V = 1;
            }
        }
    }
}

```

/* DIVU */

Assembly Code Example

```
DIVU r1,r2,r3 ; r1 is assigned r2 / r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
DIVU<.f>	a,b,c	00101bbb00000101FBBBCCCCCAAAAAA
DIVU<.f>	a,b,u6	00101bbb01000101FBBBuuuuuuAAAAAA
DIVU<.f>	b,b,s12	00101bbb10000101FBBBssssssSSSSSS
DIVU<.cc><.f>	b,b,c	00101bbb11000101FBBBCCCCC0QQQQQ
DIVU<.cc><.f>	b,b,u6	00101bbb11000101FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

```
if(cc) {  
    result = (b.h0 * c.h0) + (b.h1 * c.h1);  
    a = result.w0;  
    acc = result;  
}
```

DMULH11

Function

Multiply by D1, writing to D1

Extension Group

Double-precision Floating Point

Operation

```
D1 = D1 * (src1:src2)
dest = (D1 * (src1:src2)).high
```

Instruction Format

```
inst dest, src1, src2
```

Format Key:

dest = Destination Register

src1 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point multiply of data register D1 by the 64-bit value formed from operand 1 and operand 2. The result is stored in data register D1 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D1 = D1 * (src1:src2)          /*DMULH11*/
dest = (D1 * (src1:src2)).high
```

Assembly Code Example

```
DMULH11 r1,r2,r3 ;Double-precision floating point multiply of data
                  register ;D1 by the 64-bit value formed from operand 1 and
                  operand ;2.
```

Syntax and Encodings

Instruction Code

With Result

DMULH11<.f>	a,b,c	00110bbb00110000FBBBCCCCC	AAAAAA
DMULH11 <.f>	a,b,limm	00110bbb00110000FBBB111110	AAAAAA
DMULH11 <.f>	a,limm,c	0011011000110000F111CCCCC	AAAAAA

Without Result

DMULH11 <.f>	0,b,c	00110bbb00110000FBBBCCCCC	111110
DMULH11 <.f>	0,b,limm	00110bbb00110000FBBB111110	111110
DMULH11 <.f>	0,limm,c	0011011000110000F111CCCCC	111110

Related Instructions

[DMULH12](#) [DBNZ](#)

[DMULH21](#) [DSUBH11](#)

[DMULH22](#)

DMULH12

Multiply by D2, writing to D1

Extension Group

Double-precision Floating Point

Operation

```
D1 = D2 * (src1:src2)
dest = (D2 * (src1:src2)).high
```

Instruction Format

```
inst dest, src1, src2
```

Format Key:

```
dest    = Destination Register
src1    = Source Operand 1
src2    = Source Operand 2
```

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point multiply of data register D2 by the 64-bit value formed from operand 1 and operand 2. The result is stored in data register D1 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D1 = D2 * (src1:src2)          /*DMULH12*/
dest = (D2 * (src1:src2)).high
```

Assembly Code Example

```
DMULH12 r1,r2,r3 ;Double-precision floating point multiply of data
                  register ;D2 by the 64-bit value formed from operand 1 and
                  operand ;2.
```

Syntax and Encodings

Instruction Code

With Result

DMULH12<.f>	a,b,c	00110 bbb 00 110001 FBBB CCCCC AAAAAA
DMULH12 <.f>	a,b,limm	00110 bbb 00 110001 FBBB 111110 AAAAAA
DMULH12 <.f>	a,limm,c	00110 110 00 110001 F111 CCCCC AAAAAA

Without Result

DMULH12 <.f>	0,b,c	00110 bbb 00 110001 FBBB CCCCC 111110
DMULH12 <.f>	0,b,limm	00110 bbb 00 110001 FBBB 111110 111110
DMULH12 <.f>	0,limm,c	00110 110 00 110001 F111 CCCCC 111110

Related Instructions

[DMULH21](#) [DADDH12](#)

[DMULH22](#) [DSUBH12](#)

DMULH21

Function

Multiply by D1, writing to D2

Extension Group

Double-precision Floating Point

Operation

```
D2 = D1 * (src1:src2)
dest = (D1 * (src1:src2)).high
```

Instruction Format

```
inst dest, src1, src2
```

Format Key:

dest = Destination Register

src1 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point multiply of data register D1 by the 64-bit value formed from operand 1 and operand 2. The result is stored in data register D2 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D2 = D1 * (src1:src2)          /*DMULH21*/
dest = (D1 * (src1:src2)).high
```

Assembly Code Example

```
DMULH21 r1,r2,r3 ;Double-precision floating point multiply of data
                  register ;D1 by the 64-bit value formed from operand 1 and
                  operand ;2.
```

Syntax and Encodings

Instruction Code

With Result

DMULH21 <.f>	a,b,c	00110bbb00110010FBBBCCCCCAAAAAA
DMULH21 <.f>	a,b,limm	00110bbb00110010FBBB111110AAAAAA
DMULH21 <.f>	a,limm,c	0011011000110010F111CCCCCAAAAAA

Without Result

DMULH21 <.f>	0,b,c	00110bbb00110010FBBBCCCCC111110
DMULH21 <.f>	0,b,limm	00110bbb00110010FBBB111110111110
DMULH21 <.f>	0,limm,c	0011011000110010F111CCCCC111110

Related Instructions

DMULH12	DADDH21
DMULH22	DSUBH21

DMULH22

Function

Multiply by D2, writing to D2

Extension Group

Double-precision Floating Point

Operation

```
D2 = D2 * (src1:src2)
dest = (D2 * (src1:src2)).high
```

Instruction Format

```
inst dest, src1, src2
```

Format Key:

dest = Destination Register

src1 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point multiply of data register D2 by the 64-bit value formed from operand 1 and operand 2. The result is stored in data register D2 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D2 = D2 * (src1:src2) /*DMULH22*/
dest = (D2 * (src1:src2)).high
```


Assembly Code Example

```
DMULH22 r1,r2,r3 ;Double-precision floating point multiply of data
                  register ;D2 by the 64-bit value formed from operand 1 and
                  operand ;2.
```

Syntax and Encodings

Instruction Code

With Result

DMULH22 <.f>	a,b,c	00110bbb00110011FBBBCCCCCAAAAAA
DMULH22 <.f>	a,b,limm	00110bbb00110011FBBB111110AAAAAA
DMULH22 <.f>	a,limm,c	0011011000110011F111CCCCCAAAAAA

Without Result

DMULH22 <.f>	0,b,c	00110bbb00110011FBBBCCCCC111110
DMULH22 <.f>	0,b,limm	00110bbb00110011FBBB111110111110
DMULH22 <.f>	0,limm,c	0011011000110011F111CCCCC111110

Related Instructions

DMULH12	DADDH22
DMULH21	DSUBH22

DSUBH11

Function

Subtract from D1, writing to D1

Extension Group

Double-precision Floating Point

Operation

```
D1 = D1 - (src1:src2)
dest = (D1 - (src1:src2)).high
```

Instruction Format

```
inst dest, src1, src2
```

Format Key:

dest = Destination Register

src1 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point subtract of the 64-bit value formed by operand 1 and operand 2 from data register D1. The result is stored in data register D1 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D1 = D1 - (src1:src2) /*DSUBH11*/
dest = (D1 - (src1:src2)).high
```

Assembly Code Example

```
DSUBH11 r1,r2,r3    ;Double-precision floating point subtract of the 64-bit
                    ;value formed by operand 1 and operand 2 from data
                    ;register D1.
```

Syntax and Encodings

Instruction Code

With Result

DSUBH11 <.f> a,b,c	00110bbb00111000FBBBCCCCC	AAAAAA
DSUBH11 <.f> a,b,limm	00110bbb00111000FBBB111110	AAAAAA
DSUBH11 <.f> a,limm,c	0011011000111000F111CCCCC	AAAAAA

Without Result

DSUBH11 <.f> 0,b,c	00110bbb00111000FBBBCCCCC	111110
DSUBH11 <.f> 0,b,limm	00110bbb00111000FBBB111110	111110
DSUBH11 <.f> 0,limm,c	0011011000111000F111CCCCC	111110

Related Instructions

[DSUBH12](#) [DBNZ](#)

[DSUBH21](#) [DSUBH22](#)

DSUBH12

Function

Subtract from D2, writing to D1

Extension Group

Double-precision Floating Point

Operation

$$D1 = D2 - (src1:src2)$$

$$dest = (D2 - (src1:src2)).high$$

Instruction Format

```
inst dest, src1, src2
```

Format Key:

dest = Destination Register

src1 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point subtract of the 64-bit value formed by operand 1 and operand 2 from data register D2. The result is stored in data register D1 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D1 = D2 - (src1:src2)          /*DSUBH12*/
dest = (D2 - (src1:src2)).high
```

Assembly Code Example

```
DSUBH12 r1,r2,r3    ;Double-precision floating point subtract of the 64-bit
                    ;value formed by operand 1 and operand 2 from data
                    ;register D2.
```

Syntax and Encodings

Instruction Code

With Result

DSUBH12 <.f>	a,b,c	00110bbb00111001FBBBCCCCC	AAAAAA
DSUBH12 <.f>	a,b,limm	00110bbb00111001FBBB111110	AAAAAA
DSUBH12 <.f>	a,limm,c	0011011000111001F111CCCCC	AAAAAA

Without Result

DSUBH12 <.f>	0,b,c	00110bbb00111001FBBBCCCCC	111110
DSUBH12 <.f>	0,b,limm	00110bbb00111001FBBB111110	111110
DSUBH12 <.f>	0,limm,c	0011011000111001F111CCCCC	111110

Related Instructions

[DSUBH11](#) [DADDH12](#)

[DSUBH21](#) [DMULH12](#)

[DSUBH22](#)

DSUBH21

Function

Subtract from D1, writing to D2

Extension Group

Double-precision Floating Point

Operation

$$D2 = D1 - (src1:src2)$$

$$dest = (D1 - (src1:src2)).high$$

Instruction Format

inst dest, src1, src2

Format Key:

dest = Destination Register

src1 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point subtract of the 64-bit value formed by operand 1 and operand 2 from data register D1. The result is stored in data register D2 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
D2 = D1 - (src1:src2)          /*DSUBH21*/
dest = (D1 - (src1:src2)).high
```

Assembly Code Example

```
DSUBH21 r1,r2,r3    ;Double-precision floating point subtract of the 64-bit
                    ;value formed by operand 1 and operand 2 from data
                    ;register D1.
```

Syntax and Encodings

Instruction Code

With Result

DSUBH21 <.f> a,b,c	00110bbb00111010FBBBCCCCC	AAAAAA
DSUBH21 <.f> a,b,limm	00110bbb00111010FBBB111110	AAAAAA
DSUBH21 <.f> a,limm,c	0011011000111010F111CCCCC	AAAAAA

Without Result

DSUBH21 <.f> 0,b,c	00110bbb00111010FBBBCCCCC	111110
DSUBH21 <.f> 0,b,limm	00110bbb00111010FBBB111110	111110
DSUBH21 <.f> 0,limm,c	0011011000111010F111CCCCC	111110

Related Instructions

[DSUBH11](#) [DADDH21](#)

[DSUBH12](#) [DMULH21](#)

[DSUBH22](#)

DSUBH22

Function

Subtract from D2, writing to D2

Extension Group

Double-precision Floating Point

Operation

$$D2 = D2 - (src1:src2)$$

$$dest = (D2 - (src1:src2)).high$$

Instruction Format

inst dest, src1, src2

Format Key:

dest = Destination Register

src1 = Source Operand 1

src2 = Source Operand 2

Flag Affected (32-Bit)

Z	•	= Set if write-back result is zero	L	= Limm Data
N	•	= Set if MSB of write-back result is set		
C	•	= Set if write-back result is a NaN		
V	•	= Set if write-back result overflowed		

Description

Double-precision floating point subtract of the 64-bit value formed by operand 1 and operand 2 from data register D2. The result is stored in data register D2 and the top half of the result is returned to the destination core register. Data registers D1 and D2 are set up through the [DEXCL1](#) and [DEXCL2](#) instructions, or via auxiliary registers.

Pseudo Code

```
/*DSUBH22*/
```


Assembly Code Example

```
DSUBH22 r1,r2,r3    ;Double-precision floating point subtract of the 64-bit
                    ;value formed by operand 1 and operand 2 from data
                    ;register D2.
```

Syntax and Encodings

Instruction Code

With Result

DSUBH22 <.f>	a,b,c	00110bbb00111011FBBBCCCCC	AAAAAA
DSUBH22 <.f>	a,b,limm	00110bbb00111011FBBB111110	AAAAAA
DSUBH22 <.f>	a,limm,c	0011011000111011F111CCCCC	AAAAAA

Without Result

DSUBH22 <.f>	0,b,c	00110bbb00111011FBBBCCCCC	111110
DSUBH22 <.f>	0,b,limm	00110bbb00111011FBBB111110	111110
DSUBH22 <.f>	0,limm,c	0011011000111011F111CCCCC	111110

Related Instructions

[DSUBH11](#) [DADDH22](#)

[DSUBH12](#) [DMULH22](#)

[DSUBH21](#)

EI_S

Function

Execute Indexed

Extension Group

CODE_DENSITY

Operation

Execute instruction from table at a given index

Instruction Format

op u10

Syntax Example

EI_S u10

STATUS32 Flags Affected

Z	☐	= Dependent on instruction executed
N	☐	= Dependent on instruction executed
C	☐	= Dependent on instruction executed
V	☐	= Dependent on instruction executed

Description

The EI_S instruction can index up to 1,024 entries in the instruction table located at the EI_BASE register. Each entry in the table is typically a 32-bit encoded instruction. Because the u10 operand is a table index and each table entry is assumed to be 4 bytes, the u10 operand is scaled by a factor of 4 before being added to the table base address.

The processor jumps to the indexed table location, executes one instruction and then returns to the next instruction located sequentially after the EI_S.

Each instruction executed by the EI_S instruction from the table pointed to by EI_BASE is effectively in the execution slot of the EI_S, indicated by the STATUS32.ES bit. Instructions in the execution slot cannot be any of the following types: Bcc, BLcc, Jcc, JLcc, LPcc, RTIE, ENTER_S and LEAVE_S. All other instructions are permitted, and all operand types that are normally permitted for these instructions may be used by the instructions residing in the table. If long immediate data is used by an instruction in the table, the immediate data must immediately follow the instruction in the table.

An EI_S instruction raises an Illegal Instruction Sequence exception in the following cases:

- When JLI_S instruction is attempted in the execution slot of an EI_S instruction

- When any of the following instructions are attempted in the execution slot of an EI_S instruction:
 - Another EI_S instruction
 - Any jump or branch instruction (Bcc, BLcc, Jcc, JLcc)
 - A Loop instruction (LPcc)
 - A return from interrupt instruction (RTIE)
 - An ENTER_S or LEAVE_S instruction

**Note**

An illegal instruction sequence exception is not raised when the ENTER_S and LEAVE_S instructions have zero operands.

- When EI_S is the last instruction position of a zero-overhead loop
- When the EI_S instruction is in a branch or jump delay slot

Pseudo Code

```
BTA = PC + 2                                /* EI_S */
STATUS32.ES = 1
PC = EI_BASE + (u10 << 2)
```

Assembly Code Example

```
EI_S index    ; store the next PC in BTA, set the execution-slot status
               ;bit, and then jump to EI_BASE[index]
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
EI_S      u10    010111uuuuuuuuuu
```

ENTER_S

Function

Function Prolog Sequence

Extension Group

CODE_DENSITY

Operation

Push a collection of registers on to the stack, update stack pointer, and frame pointer, as required

Instruction Format

op u6

Syntax Example

ENTER_S {r13-r26, fp, blink}

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

This instruction executes the function prolog code, storing on the stack the set of registers defined by the 6-bit literal operand. On completion of this instruction, the stack pointer (r28) is decremented by 4S, where S is the total number of registers saved. If the frame-pointer is specified in the operand bit-vector, the stack-pointer is copied to the frame-pointer on completion of the operation.

For ENTER_S and LEAVE_S instructions, if the stack pointer is not 32-bit aligned, a misaligned access exception is always raised irrespective of the STATUS32.AD bit.

The status flags are not updated by this instruction.

The ENTER_S instruction is not allowed to appear in the delay slot of a jump or branch instruction. The processor raises an Illegal Instruction Sequence exception on any attempt to execute ENTER_S in a delay slot.



Note

An illegal instruction sequence exception is not raised when the ENTER_S instruction has zero operands.

The u6 encoding is as follows:

u[3:0]	Indicates the number of general-purpose registers to be saved on the stack, starting from r13 and counting contiguously upwards. If 32 general-purpose registers are configured, at most 14 registers (r13 to r26) can be saved. If only 16 general-purpose registers are configured, at most 3 registers (r13 to r15) can be saved. Therefore, if u[3:0] > 14 in a 32-register configuration, or if u[3:0] > 3 in a 16-register configuration, an Illegal Instruction exception is raised.
u[4]	If set to 1, r27 (frame-pointer) is saved, and on completion of the sequence, r27 is assigned the stack-pointer value (r28).
u[5]	If set to 1, the blink register (r31) is saved.

This instruction may perform as many as 16 writes to memory. These behave identically to the normal store instruction, such as:

```
ST r31, [sp, -64]
```

Pseudo Code

```

if (RGF_NUM_REGS == 32)                                /* ENTER_S */
    MaxRegs = 14
else
    MaxRegs = 3
if ((u[3:0] > MaxRegs)
    RaiseException (IllegalInstruction)

if (InDelaySlot)
    RaiseException (IllegalInstructionSequence)

S = u[3:0] + u[4] + u[5] // number of registers
W = 4
M = S * W                // stack space allocated

if (u[5] == 1)           // save blink?
{
    mem[sp - M] = r31
    M -= W
}
for (i = 0; i < u[3:0]; i++)
{
    mem[sp - M] = r[i+13] // save GPR i+13
    M -= W
}
if (u[4] == 1)           // save frame pointer?
{
    mem[sp - M] = r27
    M -= W
}

SP= sp - S*W             // allocate stack space
if (u[4] == 1)           // if frame-pointer saved
    r[27] = sp           // then set fp = sp

```

Assembly Code Example

```
ENTER_S {r13-r26, fp, blink} ;Push registers r13-r26, fp and blink to
                               ;the stack, updating the stack pointer.
                               ;Afterwards, the frame pointer assigns the
                               ;new stack pointer value.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
ENTER_S    u6    110000UU111uuuu0
```

EX

Function

Atomic Exchange

Extension Group

BASELINE

Operation

temp = b ; b = *c; *c = temp;

Instruction Format

op b, c

Syntax Example

EX<.di> b,[c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

An atomic exchange operation, EX, is provided as a primitive for multiprocessor synchronization allowing the creation of semaphores in shared memory.

Two forms are provided: an uncached form (using the .DI directive) for synchronization between multiple processors in systems without cache-coherent shared memory, and a cached form for synchronization between processes on a single-processor system, or on a multiprocessor system with cache-coherent shared memory. The cached EX is completely handled by the core and does not depend on the AXI subsystem. The uncached EX relies on locked bus transactions that must be supported in the memory subsystem. Note that the locked bus transactions is supported in AXI3 but it is not supported in AXI4. For more information about this instruction, see the [Atomic Memory Operations](#) section.

The EX instruction exchanges the contents of the specified memory location with the contents of the specified register. This operation is atomic and the memory system ensures that the memory read and memory write cannot be separated by interrupts or by memory accesses from another source. The memory location must be 32-bit aligned (address bits [1:0] = 0). Thus, the EX-related accesses to data cache never cross a cache line boundary. Even if the data memory alignment checks are disabled

(STATUS32.AD bit is set to 1), an EX instruction to an unaligned memory address always raises an EV_Misaligned (Misaligned Data Access) exception.

The status flags are not updated with this instruction.

An immediate value is not permitted to be the destination of the exchange instruction. Using the long immediate indicator in the destination field, B=0x3E, raises a [Illegal Instruction](#) exception.



When used in conjunction with an MMU or MPU, both the read and write permissions must be set for EX to operate without causing a protection violation exception.

Pseudo Code

```
int32_t                                     /* EX */
swap( int32_t new_value, int32_t *word ) {
    int32_t  old_value;
    atomic {
        old_value = *word;
        *word     = new_value;
    }
    return( old_value );
}
```

Assembly Code Example

In this example, the processor attempts to get access to a shared resource by testing a semaphore against values 0 and 1.

- If the returned value is a 0, the resource is free and this device is now the owner.
- If the returned value is a 1, the resource is busy and the processor must wait till a 0 is returned.

The value 1 is always written to SEMAPHORE_ADDR. All processes trying to own the semaphore must write the same value.

The value at SEMAPHORE_ADDR must not be used to determine the current owner of the semaphore.

Example 10-1 To obtain a semaphore using EX

```
wait_for_resource:                ; Obtain a semaphore using EX
    MOV R2, 0x00000001            ; 1 => semaphore is owned

wfr1:
    EX  R2, [SEMAPHORE_ADDR]      ; exchange r2 and semaphore
    CMP_S R2, 0                   ; see if we own the semaphore
    BNE wfr1                      ; wait for resource to free
```

Example 10-2 To Release Semaphore using ST

```
release_resource:                ; Release Semaphore using ST
MOV R2, 0x00000000              ; indicates semaphore is free
ST  R2, [SEMAPHORE_ADDR]        ; release semaphore
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
EX<.di>	b, [c]	00100bbb00101111DBBBCCCCCC001100
EX<.di>	b, [u6]	00100bbb01101111DBBBuuuuuu001100

EXTB

Function

Zero Extend Byte

Extension Group

BASELINE

Operation

$b = c \& 0x000000FF;$

Instruction Format

op b, c

Syntax Example

EXTB<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always Zero
C		= Unchanged
V		= Unchanged

Description

Zero extend the byte value in the source operand (c) and write the result into the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src & 0xFF                                /* EXTB */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
EXTB r3,r0    ; Zero extend the bottom 8bits of r0 and write result to r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

EXTB<.f>	b,c	00100bbb00101111FBBBCCCCC000111
EXTB<.f>	b,u6	00100bbb01101111FBBBuuuuuu000111
EXTB_S	b,c	01111bbbccc01111

EXTH

Function

Zero Extend Half-word (16-bit)

Extension Group

BASELINE

Operation

$b = c \ \& \ 0x0000FFFF;$

Instruction Format

op b, c

Syntax Example

EXTH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always Zero
C		= Unchanged
V		= Unchanged

Description

Zero extend the 16-bit half-word value in the source operand (c) and write the result into the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src & 0xFFFF          /* EXTH */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
EXTH r3,r0    ; Zero extend the bottom 16 bits of r0 and write result to r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

EXTH<.f>	b, c	00100bbb00101111FBBBCCCCC001000
EXTH<.f>	b, u6	00100bbb01101111FBBBuuuuuu001000
EXTH_S	b, c	01111bbbccc10000

FCVT32

Function

Convert between two 32-bit data formats: single-precision float, signed or unsigned integer.

Alias

FS2INT, FS2UINT, FS2INT_RZ, FS2UINT_RZ, FINT2S, FUINT2S

Extension Group

FPU,

Operation

```
a = (int) b;
a = (float) b;
a = (uint)b;
```

Instruction Format

op a, b, u6

Syntax Example

FCVT32 a,b,u6

STATUS32 Flags Affected

STATUS32 Flags

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	- Set on floating-point to integer conversion if: - Floating-point number is NaN - Floating-point number is infinity - Conversion result is outside the range of the integer data type (range is [0,65535] for unsigned short). - or - If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).

DZ		= Unchanged
OF	•	Set when conversion from single precision to half precision results in a number too large to be represented (that is: the result is finite and has a magnitude that exceeds the maximum normalized number).
UF		= Unchanged
IX	•	Set when conversion from single precision to half precision results in a number too small to be represented (that is: the result after rounding has a magnitude less than the minimum normalized number), and it is inexact. or Set when conversion from single precision to half precision has a result that is rounded (that is: the result after rounding is different from the infinitely precise value before rounding).

Description

This instruction converts a 32-bit data format into another 32-bit data format. The type of conversion is determined by the `u6` operand. Depending on the value of `u6`, the processor does the following conversions: Bits 4 to 31 of the source 2 value passed to these instructions are ignored.

Instruction	u6 value	Description
FS2INT	FCVT32 a, b, 0b000011	Single-precision float to signed integer
FS2INT_RZ	FCVT32 a, b, 0b001011	Single-precision float to signed integer; round to zero
FINT2S	FCVT32 a, b, 0b000010	Signed integer to single-precision float
FS2UINT	FCVT32 a, b, 0b000001	Single-precision float to unsigned integer
FS2UINT_RZ	FCVT32 a, b, 0b001001	Single-precision float to unsigned integer; round to zero
FUINT2S	FCVT32 a, b, 0b000000	Unsigned integer to single-precision float

Pseudo Code

```
a = (float) b; /* FCVT32 */
```

Assembly Code Example

```
FCVT32 r2,r3, u6 ; conversion from r2 to r3
```


Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FCVT32	a, b, c	00110bbb000010000BBBCCCCCAAAAAA
FCVT32	a, b, u6	00110bbb010010000BBBuuuuuuAAAAAA
FCVT32	b, b, s12	00110bbb100010000BBBssssssSSSSSS
FCVT32<.cc>	b, b, c	00110bbb110010000BBBCCCCC0QQQQQ
FCVT32<.cc>	b, b, u6	00110bbb110010000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FFS

Function

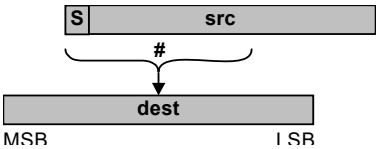
Find First Set

Extension Group

BITSCAN_OPTION

Operation

dest ← bit position of first 1 in source operand, from bit 0 upwards



Instruction Format

op b,c

Syntax Example

FFS<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Returns the bit position of the least-significant (that is, lowest numbered) non-zero bit in the source operand. Examples of returned values are shown in the following table. The '-' symbol signifies any hexadecimal digit:

Operand Value	Result	Z	N
0x-----1	0	0	0
0x-----2	1	0	0
0x-----4	2	0	0

Operand Value	Result	Z	N
0x-----8	3	0	0
:	:	:	:
0x40000000	30	0	0
0x80000000	31	0	1
0x00000000	31	1	0

Pseudo Code

```
s = 0                                /* FFS */

while (src && (((src >> s) & 1) == 0))
    ++s;

dest = (src == 0) ? 31 : s;

if (F == 1) then {
    Z_flag = (src == 0) ? 1 : 0;
    N_flag = src[31];
}
```

Assembly Code Example

```
FFS r1,r2    ; Find least-significant non-zero bit in r2 write result
              into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
FFS<.f> b,c	00101	bbb00101111FBBBCCCCC010010
FFS<.f> b,u6	00101	bbb01101111FBBBuuuuuu010010

FLAG

Function

Set Flags

Extension Group

BASELINE

Operation

if (cc) assign bits from operand to STATUS32

Instruction Format

op c

Syntax Example

FLAG<.cc> c

STATUS32 Flags Affected

Flag		Mode	Source of operand
IE			Unchanged
US	•	Kernel	Bit 20 of Source Operand
AD	•	Kernel	Bit 19 of Source Operand
RB[3:0]			Unchanged
SC	•	Kernel	Bit 14 of Source Operand
DZ	•	Kernel	Bit 13 of Source Operand, if DIV_REM_OPTION is configured
L			Unchanged
Z	•	Any	Bit 11 of Source Operand
N	•	Any	Bit 10 of Source Operand
C	•	Any	Bit 9 of Source Operand
V	•	Any	Bit 8 of Source Operand
U			Unchanged
DE			Unchanged
AE			Unchanged

E[3:0]	•	Kernel	Bit [4:1] of Source Operand
H	•	Kernel	Bit 0 of Source Operand (If set, ignore all other flags states)
H	•	Secure Kernel	Bit 0 of Source Operand (If set, ignore all other flags states)

Description

The contents of the source operand (c) are used to set the condition code and processor control flags held in the processor status registers.

The format of the source operand is identical to the format used by the STATUS32 register (auxiliary address 0x0A).

Bit 14 of the source operand relates to the stack-checking enable, bit 13 relates to the divide-by-zero exception enable, bits [11:8] of the source operand relate to the condition codes, and bit [0] relates to the halt flag. All other bits are ignored. If you attempt to set STATUS32.H to 1 in the normal kernel mode, a Privilege Violation is generated. Attempts to set the H bit using FLAG in both normal user mode and secure user mode are ignored.

If the H flag is set (halt processor flag), all other flag states are ignored and are not updated.

The arithmetic flags in the STATUS32 register (auxiliary address 0x0A) are always updated by the FLAG instruction. The flag setting field, F, is always encoded as 0 for this instruction.

Only the Z, N, C and V flags are modified by a FLAG instruction in user mode; all other flags remain unchanged.

The FLAG instruction is serializing – ensuring that no further instructions can be completed before all flag updates take effect.

Bits U, DE, AE, RB, IE, and L in the STATUS32 register are not modified by the FLAG instruction in any operating mode. These are updated when the processor changes state because of branches, interrupts, or exceptions, and when the processor returns from interrupts or exception by executing the RTIE instruction.

Pseudo Code

```
FlushPipe()
if ((src[0] == 1) && (STATUS32[7] == 0)) then
    if (HAS_INTERRUPTS > 0)
        STATUS32[0] = 1
        Halt()
    else
        STATUS32[11:8] = src[11:8]
        if (STATUS32[7] == 0 && HAS_INTERRUPTS > 0) /*in kernel mode*/
            STATUS32[4:1] = src[4:1]
            if (DIV_REM_OPTION == 1)
                STATUS32[13] = src[13]
            if (STACK_CHECKING == 1)
                STATUS32[14] = src[14]
            if (LL64_OPTION == 1)
                STATUS32[19] = src[19]
            STATUS32[20] = src[20]
/* FLAG
when
SecureShield is not
configured
*/
```

```

FlushPipe()
if ((src[0] == 1) && (STATUS32.U == 0)) then
    if (STATUS32.S == 1) then
STATUS32.H = 1
Halt()
    else
        /* in non-secure mode */
        exception(EV_PrivilegeV, ECR = 0x071004)
else
    STATUS32[11:8] = src[11:8]
    /*ZNCV, can be updated in any mode*/
if (STATUS32.U == 0)
/*in kernel mode*/
    if (SEC_MODES_OPTION == 1)
        /*SecureShield is present*/
        if (STATUS32.S == 1)
            /*in secure mode*/
            STATUS32.E = src[4:1]
            /*E[3:0]*/
        else /*SecureShield is not present*/
            STATUS32.E = src[4:1]
            /*E[3:0]*/

    STATUS32.AE = src[5]
    /*AE*/
    if (DIV_REM_OPTION == 1)
        STATUS32.DZ = src[13]
        /*DZ*/
    if (STACK_CHECKING == 1)
        STATUS32.SC = src[14]
        /*SC*/
if (SEC_MODES_OPTION == 0)
/*SecureShield is not present*/
    if (LL64_OPTION == 1)
        STATUS32.AD = src[19]
        /*AD*/
    STATUS32.US = src[20]
    /*US*/

```

/* FLAG
when
SecureShield is
configured

Assembly Code Example

```

FLAG 1          ; Halt processor (other flags
                 ; not updated)
FLAG 6          ; Set the Interrupt threshold to
                 ; 3. Any interrupt with equal or
                 ; higher priority than 3 is
                 ; serviced if interrupts are
                 ; enabled.

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

FLAG	c	00100rrr001010010RRRCCCCCRRRRRR
FLAG	u6	00100rrr011010010RRRuuuuuuRRRRRR
FLAG	s12	00100rrr101010010RRRssssssSSSSSS
FLAG<.cc>	c	00100rrr111010010RRRCCCCC0QQQQQ
FLAG<.cc>	u6	00100rrr111010010RRRuuuuuu1QQQQQ

FLS

Function

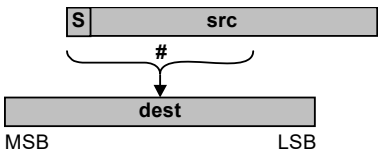
Find Last Set

Extension Group

BITSCAN_OPTION

Operation

dest ← bit position of last 1 in source operand, from bit 0 upwards



Instruction Format

op b,c

Syntax Example

FLS<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Returns the bit position of the most-significant non-zero bit in the source operand. If the source operand is 0 (zero), the value 0 (zero) is returned. Examples of returned values are shown in the following table. The '-' symbol signifies any hexadecimal digit:

Operand Value	Result	Z	N
Any value greater than 0x80000000 (inclusive)	31	0	1
Any value between 0x40000000 (inclusive) and 0x80000000	30	0	0

Operand Value	Result	Z	N
Any value between 0x20000000 (inclusive) and 0x40000000	29	0	0
:	:	:	:
Any value between 0x00000004 (inclusive) and 0x00000008	2	0	0
Any value between 0x00000002 (inclusive) and 0x00000004	1	0	0
0x00000001	0	0	0
0x00000000	0	1	0

Pseudo Code

```

s = 31                                /* FLS */

while (src && (((src >> s) & 1) == 0))
    --s;

dest = s;

if (F == 1) then {
    Z_flag = (src == 0) ? 1 : 0;
    N_flag = src[31];
}

```

Assembly Code Example

```

FLS r1,r2    ; Find most-significant non-zero bit in r2 and write
              ;result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```

FLS<.f>    b,c    00101bbb00101111FBBBCCCCC010011
FLS<.f>    b,u6   00101bbb01101111FBBBuuuuuu010011

```

FSABS

Function

Single-precision floating-point absolute operation.

Alias

BCLR r1, r3, 0x1F

Extension Group

BASELINE

Operation

$b = (\text{float}) \text{ abs}(c);$

Instruction Format

op b, c

Syntax Example

FSABS b, c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	☐	= Unchanged
DZ	☐	= Unchanged
OF	☐	= Unchanged
UF	☐	= Unchanged
IX	☐	= Unchanged

Description

This instruction returns the result of the absolute of the source operand, *c*, into the destination register, *b*. As floating-point numbers are signed magnitude representations, the absolute operation involves clearing the sign bit of the source operand.

Pseudo Code

```
if cc==true then                                /* FSABS */
dest = abs(src)
```

Assembly Code Example

```
FSABS r1, r2      ; return the absolute of r2 into r1
```

Syntax and Encoding

FSABS is supported using an assembly alias. For syntax and encoding information, refer to the BCLR instruction used in the following format:

```
BCLR r1, r2, 0x1F
```

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

FSADD

Function

Single-precision floating-point addition

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {  
    a = (float) (b + c);  
}
```

Instruction Format

op a,b,c

Syntax Example

FSADD a,b,c

FSADD<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	Set when the operands are in the following combinations: b= +infinity and c = -infinity or b= -infinity and c = +infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The sum of b and c operands is stored in the destination register.

Pseudo Code

```
if(cc) {                                     /* FSADD */
    a = (float) (b + c);
}
```

Assembly Code Example

```
FSADD r1,r2,r3    ; floating-point addition of r2 and r3 and the
                  ; result is stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
FSADD	a,b,c	00110bbb00000010BBBCCCCCAAAAAA
FSADD	a,b,u6	00110bbb010000010BBBuuuuuuAAAAAA
FSADD	b,b,s12	00110bbb100000010BBBsssssssSSSSSS
FSADD<.cc>	b,b,c	00110bbb110000010BBBCCCCC0QQQQQ

FSADD<.cc> b,b,u6 00110bbb110000010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSCMP

Function

Single-precision floating-point comparison

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {
    compare(b , c);
}
```

Instruction Format

op b,c

Syntax Example

```
FSCMP b,c
FSCMP<.cc> b,c
```

STATUS32 Flags Affected

Z	•	= Set when b and c are equal
N	•	= Set when b is less than c
C	•	= Set when b is less than c
V	•	Set when the comparison result is unordered. If this bit is set, the Z, N, C flags are cleared.

FPU_STATUS Flags

IV	•	Set when either of the inputs is a signaling NaN and FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The b and c operands are compared and the core flags are updated.



Note

If the source operands have a magnitude of zero (0), the Z flag is always updated irrespective of the sign of the source operands.

Pseudo Code

```
if(cc) { /* FSCMP */
    Z_flag = (src1 == src2) && (src1 != NaN) && (src2 != NaN);
    N_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    C_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    V_flag = (src1 == NaN) || (src2 == NaN);
    FPU_STATUS.IV = (src1 == sNaN) || (src2 == sNaN);
}
```

Assembly Code Example

```
FSCMP r2,r3      ; floating-point comparison of r2 and r3 and the
                  ; core flags are updated
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

FSCMP	b, c	00110bbb00000111BBBCCCCC000000
FSCMP	b, u6	00110bbb010000111BBBuuuuuu000000
FSCMP	b, s12	00110bbb100000111BBBssssssSSSSSS
FSCMP<.cc>	b, c	00110bbb110000111BBBCCCCC0QQQQQ
FSCMP<.cc>	b, u6	00110bbb110000111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSCMPF

Function

Single-precision floating-point comparison operation with IEEE 754 flag generation. This instruction is similar to the FSCMP instruction in cases when either of the instruction operands is a signaling NaN. The FSCMPF instruction updates the invalid flag (FPU_STATUS.IV) when either of the operands is a quiet or signaling NaN, whereas, the FSCMP instruction updates the invalid flag (FPU_STATUS.IV) only when either of the operands is a signaling NaN.

Extension Group

FPU

Operation

```
if (cc) {
    compare(b , c);
}
```

Instruction Format

op b,c

Syntax Example

```
FSCMPF b,c
FSCMPF<.cc> b,c
```

STATUS32 Flags Affected

Z	•	= Set when b and c are equal
N	•	= Set when b is less than c
C	•	= Set when b is less than c
V	•	= Set when the comparison result is unordered. If this bit is set, the Z, N, C flags are cleared

FPU_STATUS Flags

IV	•	Set when either of the inputs is a quiet or signaling NaN and FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The b and c operands are compared and the core flags are updated.



Note

If the source operands have a magnitude of zero (0), the Z flag is always updated irrespective of the sign of the source operands.

Pseudo Code

```
if(cc) {
    Z_flag = (src1 == src2) && (src1 != NaN) && (src2 != NaN);
    N_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    C_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    V_flag = (src1 == NaN) || (src2 == NaN);
    FPU_STATUS.IVF = V_flag;
}
```

Assembly Code Example

```
FSCMPF r2,r3      ; floating-point comparison of r2 and r3 and update flags
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

FSCMPF	b, c	00110bbb000001001BBBCCCCC000000
FSCMPF	b, u6	00110bbb010001001BBBuuuuuu000000
FSCMPF	b, s12	00110bbb100001001BBBssssssSSSSSS

FSCMPF<.cc>	b, c	00110bbb110001001BBBCCCCC0QQQQQ
FSCMPF<.cc>	b, u6	00110bbb110001001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSDIV

Function

Single-precision floating-point division

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {
    a = (float) (b / c);
}
```

Instruction Format

op a,b,c

Syntax Example

FSDIV a,b,c

FSDIV<cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	Set when the operands are in the following combinations: 0/0 or infinity/infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Set when the divisor is zero
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The b operand is divided by the c operand, and the result is stored in the destination register.

Pseudo Code

```
if(cc) {                                     /* FSDIV */
    a = (float) (b / c);
}
```

Assembly Code Example

```
FSDIV r1,r2,r3    ; floating-point division of r2 by r3 and the
                  ; result is stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FSDIV	a, b, c	00110bbb000001110BBBCCCCCAAAAAA
FSDIV	a, b, u6	00110bbb010001110BBBuuuuuuAAAAAA
FSDIV	b, b, s12	00110bbb100000010BBBsssssssSSSSSS
FSDIV<.cc>	b, b, c	00110bbb110001110BBBCCCCC0QQQQQ

FSDIV<.cc> b,b,u6 00110bbb110001110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSMADD

Function

Single-precision floating-point fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply-add operation performed in one step, with a single rounding.

Extension Group

(HAS_FPU == 1) && (FPU_FMA_OPTION == 1)

Operation

```
if (cc) {
    a = (float) (acc + (b * c));
}
```

Instruction Format

op a,b,c

Syntax Example

FSMADD a,b,c

FSMADD<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	- Set when the b and c operands are in the following combinations: $0 * \text{infinity}$ or $\text{infinity} * 0$ - Set when the a, b, and c operands are in the following combinations: $(b * c) = +\text{infinity}$ and $a = -\text{infinity}$ or $(b * c) = -\text{infinity}$ and $a = +\text{infinity}$ - If any of the input operands is a signaling NaN when <code>FPU_CTRL.IVE==0</code> (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded after the accumulation

Description

The product of b and c operands is computed, added to the accumulator, and the resulting sum is rounded and stored in the destination register.



Note

This instruction uses the accumulator as described in “[Extension Core Registers](#)”.

Pseudo Code

```
if(cc) {
    a = (float) (acc + (b * c));
}
```

/* FSMADD */

Assembly Code Example

```
FSMADD r1,r2,r3    ; floating-point multiplication of r2 and r3 and the
                   ; product is accumulated and the sum stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FSMADD	a, b, c	00110bbb000001010BBBCCCCCAAAAAA
FSMADD	a, b, u6	00110bbb010001010BBBuuuuuuAAAAAA
FSMADD	b, b, s12	00110bbb100001010BBBssssssSSSSSS
FSMADD<.cc>	b, b, c	00110bbb110001010BBBCCCCC0QQQQQ
FSMADD<.cc>	b, b, u6	00110bbb110001010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSMSUB

Function

Single-precision floating-point fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply-subtract operation performed in one step, with a single rounding.

Extension Group

(HAS_FPU == 1) && (FPU_FMA_OPTION == 1)

Operation

```
if (cc) {
    a = (float) (acc - (b * c));
}
```

Instruction Format

op a,b,c

Syntax Example

FSMSUB a,b,c

FSMSUB<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	- Set when the b and c operands are in the following combinations: $0 * \text{infinity}$ or $\text{infinity} * 0$ - Set when the a, b, and c operands are in the following combinations: $(b * c) = +\text{infinity}$ and $a = +\text{infinity}$ or $(b * c) = -\text{infinity}$ and $a = -\text{infinity}$ - If any of the input operands is a signaling NaN when <code>FPU_CTRL.IVE==0</code> (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The product of b and c operands is computed, subtracted from the accumulator; the result is rounded and stored in the destination register.

**Note**

This instruction uses the accumulator as described in “[Extension Core Registers](#)”.

Pseudo Code

```
if(cc) {
    a = (float) (acc - (b * c));
}
```

/* FSMSUB */

Assembly Code Example

```
FSMSUB r1,r2,r3    ; floating-point multiplication of r2 and r3 and the
                   ; product is subtracted from the accumulator and result is
                   ; stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FSMSUB	a, b, c	00110bbb000001100BBBCCCCCAAAAAA
FSMSUB	a, b, u6	00110bbb010001100BBBuuuuuuAAAAAA
FSMSUB	b, b, s12	00110bbb100001100BBBssssssSSSSSS
FSMSUB<.cc>	b, b, c	00110bbb110001100BBBCCCCC0QQQQQ
FSMSUB<.cc>	b, b, u6	00110bbb110001100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSMUL

Function

Single-precision floating-point multiply

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {
    a = (float) (b * c);
}
```

Instruction Format

op a,b,c

Syntax Example

FSMUL a,b,c

FSMUL<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	- Set when (0*infinity) or (infinity*0) operations occur - Or - If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	= Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The b and c operands are multiplied and the product is stored in the destination register.

Pseudo Code

```
if(cc) {
    a = (float) (b * c);
}
```

/* FSMUL */

Assembly Code Example

```
FSMUL r1,r2,r3    ; floating-point multiplication of r2 and r3 and the
                  ; result is stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FSMUL<.f>	a, b, c	00110bbb00000000BBBCCCCCAAAAAA
FSMUL<.f>	a, b, u6	00110bbb010000000BBBuuuuuuAAAAAA
FSMUL<.f>	b, b, s12	00110bbb100000000BBBsssssssSSSSSS
FSMUL<.cc>	b, b, c	00110bbb110000000BBBCCCCC0QQQQQ
FSMUL<.cc>	b, b, u6	00110bbb110000000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSNEG

Function

Single-precision floating-point negate operation.

Alias

BXOR r1, r2, 0x1F

Extension Group

BASELINE

Operation

$b = (\text{float}) \text{ neg}(c);$

Instruction Format

op b, c

Syntax Example

FSNEG b, c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	☐	= Unchanged
DZ	☐	= Unchanged
OF	☐	= Unchanged
UF	☐	= Unchanged
IX	☐	= Unchanged

Description

This instruction returns the result of the negation of the source operand, *c*, into the destination register, *b*. As floating-point numbers are signed magnitude representations, the negate operation involves inverting the sign bit of the source operand.

Pseudo Code

```
if cc==true then                                /* FSNEG */
dest = neg(src)
```

Assembly Code Example

```
FSNEG r1, r2      ; return the negation of r2 into r1.
```

Syntax and Encoding

FSNEG is supported using an assembly alias. For syntax and encoding information, refer to the BXOR instruction used in the following format:

```
BXOR r1, r2, 0x1F
```

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

FSSQRT

Function

Single-precision floating-point square-root

Extension Group

(HAS_FPU == 1) && (FPU_DIV_OPTION == 1)

Operation

```
if (cc) {
    b = (float) sqrt(c);
}
```

Instruction Format

op b,c

Syntax Example

FSSQRT b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	Set when the c operand is less then -zero Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	= Unchanged
OF	= Unchanged
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The b operand contains the square-root of the c operand.

Pseudo Code

```
b = (float) sqrt(c);          /* FSSQRT */
```

Assembly Code Example

```
FSSQRT r2,r3      ; floating-point square-root of r3 is stored in r2
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
FSSQRT	b, c	00110bbb001011110BBBCCCCC000000
FSSQRT	b, u6	00110bbb011011110BBBuuuuuu000000

FSSUB

Function

Single-precision floating-point subtraction

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {  
    a = (float) (b - c);  
}
```

Instruction Format

op a,b,c

Syntax Example

FSSUB a,b,c

FSSUB<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The difference of b and c operands is stored in the destination register.

Pseudo Code

```
if(cc) {                                     /* FSSUB */
    a = (float) (b - c);
}
```

Assembly Code Example

```
FSSUB r1,r2,r3    ; floating-point subtraction of r3 from r2; and the
                  ; result is stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FSSUB	a, b, c	00110bbb000000100BBBCCCCCAAAAAA
FSSUB	a, b, u6	00110bbb010000100BBBuuuuuuAAAAAA
FSSUB	b, b, s12	00110bbb100000100BBBsssssssSSSSSS
FSSUB<.cc>	b, b, c	00110bbb110000100BBBCCCCC0QQQQQ
FSSUB<.cc>	b, b, u6	00110bbb110000100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

J

Function

Jump unconditionally

Extension Group

BASELINE

Operation

PC = c;

Instruction Format

op c

Syntax Example

J [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Use this instruction to jump to the new program counter address that is specified as the absolute address in the source operand (c). Jump instructions can target any address within the full memory address map, but the target address is always 16-bit aligned. Because the delay slot mode controls the execution of the instruction which is in the delay slot, the instruction must never be the target of any branch or jump instruction.



Caution

The J and J_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

Returning from an branch-and-link or jump-and-link is accomplished by jumping to the contents of the BLINK register (see [Link Registers, ILINK \(r29\), BLINK \(r31\)](#)), using the J [BLINK] instruction.

The particular delay slot modes for the jump instructions are:

Delay Slot Mode	Description
J	Never execute next instruction
J_S	Never execute next instruction
J.D	Always execute next instruction
J_S.D	Always execute next instruction

[Table 7-36](#) on page [332](#) provides more information about the delay slot modes, .d.
[Table 7-7](#) on page [308](#) provides information about the condition codes, cc.

Description

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Indexed jump or execute instructions (JLI_s and EI_S)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```
if N==1 then                                /* J */
    DelaySlot(nPC)
PC = src
```

Assembly Code Example

```
J [r1]                                     ; jump to address in r1
NOP
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
J	[c]	00100RRR00100000RRRRCCCCCRRRRRR
J	u6	00100RRR01100000RRRRuuuuuuRRRRRR

J	s12	00100RRR10100000RRRRssssssSSSSSS
J.D	[c]	00100RRR00100001RRRRCCCCCRRRRRR
J.D	u6	00100RRR01100001RRRRuuuuuuRRRRRR
J.D	s12	00100RRR10100001RRRRssssssSSSSSS
J_S	[b]	01111bbb00000000
J_S.D	[b]	01111bbb00100000
J_S	[BLINK]	0111111011100000
J_S.D	[BLINK]	0111111111100000

Jcc

Function

Jump Conditionally

Extension Group

BASELINE

Operation

if (cc) PC = c;

Instruction Format

op c

Syntax Example

Jcc [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

If the specified condition is met (cc = true), the program execution is resumed from the new program counter address that is specified as the absolute address in the source operand (c). Jump instructions can target any address within the full memory address map, but the target address is always 16-bit aligned. Because the delay slot mode controls the execution of the instruction which is in the delay slot, the instruction must never be the target of any branch or jump instruction.



Caution

The Jcc and J_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The Jcc.D instruction may not use a long-immediate value as the target address operand.

Returning from an branch-and-link or jump-and-link is accomplished by jumping to the contents of the BLINK register (see [Link Registers, ILINK \(r29\), BLINK \(r31\)](#)), using the Jcc [BLINK] instruction.

The particular delay slot modes for the jump instructions are:

Delay Slot Mode	Description
JEQ_S	Only execute next instruction when <i>not</i> branching
JNE_S	Only execute next instruction when <i>not</i> branching
Jcc.D	Always execute next instruction

[Table 7-36](#) on page 332 provides more information about the delay slot modes, .d.

[Table 7-7](#) on page 308 provides information about the condition codes, cc.

Description

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```

if cc==true then                                /* Jcc */
  if N==1 then
    DelaySlot(nPC)
    PC = src
  else
    PC = nPC

```

Assembly Code Example

```

JEQ [r1]    ; jump to address in r1 if the Z flag is set
NOP

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

Jcc	[c]	00100RRR11100000RRRCCCCC0QQQQQ
Jcc	u6	00100RRR11100000RRRuuuuuu1QQQQQ

Jcc.D	[c]	00100RRR11100001RRRRCCCCC0QQQQQ
Jcc.D	u6	00100RRR11100001RRRRuuuuuu1QQQQQ
JEQ_S	[BLINK]	0111110011100000
JNE_S	[BLINK]	0111110111100000

JL

Function

Jump and Link unconditionally

Extension Group

BASELINE

Operation

PC = c; BLINK = NEXT_PC;

Instruction Format

op c

Syntax Example

JL [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

When this instruction is executed, the program execution is resumed from the new program counter address that is specified as the absolute address in the source operand (c). Jump and link instructions can have any target address within the full memory address map, but the target address is 16-bit aligned. In parallel, the program counter address (PC) that immediately follows the jump instruction is written into the BLINK register (R31). Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction.



Caution

The JL and JL_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The JL.D instruction may not use a long-immediate value as the target address operand.

The particular delay slot modes for the jump and link instructions are:

Delay Slot Mode	Description
JL	Never execute next instruction
JL_S	Never execute next instruction
JL.D	Always execute next instruction
JL_S.D	Always execute next instruction

[Table 7-36](#) on page [332](#) provides more information about the delay slot modes, .d.

[Table 7-7](#) on page [308](#) provides more information about the condition codes, cc.

An [Illegal Instruction Sequence](#) exception is raised if an executed delay slot contains any of the following:

- ❑ Another jump or branch instruction
- ❑ Conditional loop instruction (LPcc)
- ❑ Indexed jump or execute instructions (JLI_s and EI_S)
- ❑ Return from interrupt ([RTIE](#))
- ❑ Any instruction with long-immediate data as a source operand

Pseudo Code

```
if N==1 then                                /* JL */
    BLINK = dPC
    DelaySlot(nPC)
else
    BLINK = nPC
PC = src;
```

Assembly Code Example

```
JL [r1]      ; jump and link to address
              ; in r1 and store the return address in BLINK
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
JL	[c]	00100RRR00100010RRRCCCCCRRRRRR
JL	u6	00100RRR01100010RRRuuuuuuRRRRRR

JL	s12	00100RRR10100010RRRRssssssSSSSSS
JL.D	[c]	00100RRR00100011RRRRCCCCCRRRRRR
JL.D	u6	00100RRR01100011RRRRuuuuuuRRRRRR
JL.D	s12	00100RRR10100011RRRRssssssSSSSSS
JL_S	[b]	01111bbb01000000
JL_S.D	[b]	01111bbb01100000

JLcc

Function

Jump and Link Conditionally

Extension Group

BASELINE

Operation

if (cc) {PC = c; BLINK = NEXT_PC;}

Instruction Format

op c

Syntax Example

JLcc [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

If the specified condition is met (cc = true), the program execution is resumed from the new program counter address that is specified as the absolute address in the source operand (c). Jump and link instructions can have any target address within the full memory address map, but the target address is 16-bit aligned. In parallel, the program counter address (PC) that immediately follows the jump instruction is written into the BLINK register (R31). Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction.



Caution

The JLcc and JL_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The JLcc.D instruction may not use a long-immediate value as the target address operand

The particular delay slot modes for the jump and link instructions are:

Table 10-2 Delay Slot Modes for the Jump and Link Instructions

Delay Slot Mode	Description
JLcc	Only execute next instruction when <i>not</i> branching
JL	Never execute next instruction
JL_S	Never execute next instruction
JLcc.D	Always execute next instruction
JL.D	Always execute next instruction
JL_S.D	Always execute next instruction

[Table 7-36](#) on page [332](#) provides more information about the delay slot modes, .d.

[Table 7-7](#) on page [308](#) provides more information about the condition codes, cc.

An [Illegal Instruction Sequence](#) exception is raised if an executed delay slot contains any of the following:

- ❑ Another jump or branch instruction
- ❑ Conditional loop instruction (LPcc)
- ❑ Return from interrupt ([RTIE](#))
- ❑ Any instruction with long-immediate data as a source operand

Pseudo Code

```

if cc==true then                                /* JLcc */
  if N==1 then
    BLINK = dPC
    DelaySlot(nPC)
  else
    BLINK = nPC
  PC = src
else
  PC = nPC

```

Assembly Code Example

```

JLEQ [r1]    ; if the Z flag is set then jump and link to address
              ; in r1 and store the return address in BLINK

```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code

JLcc	[c]	00100RRR11100010RRRRCCCCC0QQQQQ
JLcc	u6	00100RRR11100010RRRRuuuuuu1QQQQQ
JLcc.D	[c]	00100RRR11100011RRRRCCCCC0QQQQQ
JLcc.D	u6	00100RRR11100011RRRRuuuuuu1QQQQQ

JLI_S

Function

Jump and Link Indexed

Extension Group

CODE_DENSITY

Operation

$BLINK = next\_PC; PC = JLI\_BASE + (u10 \ll 2);$

Instruction Format

op u10

Syntax Example

JLI_S u10

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The address of the next instruction is stored in the link register BLINK (R31). Program execution then resumes from the location given by the contents of the JLI_BASE auxiliary register plus the unsigned 10-bit index operand shifted left by two bit positions.

The JLI_S instruction can index up to 1,024 entries in the jump table defined by the JLI_BASE register. Each entry in the table is typically a 32-bit encoded relative unconditional branch instruction. Because the u10 operand is a table index and each table entry is assumed to be 4 bytes, the u10 operand is scaled by a factor of 4 before being added to the table base address.

Pseudo Code

```
BLINK = PC + 2          /* JLI_S */
PC = JLI_BASE + (u10 << 2)
```

Assembly Code Example

```
JLI_S index      ; store the return address in BLINK, and  
                  ; then jump to JLI_BASE[index]
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
JLI_S u10 010110uuuuuuuuuu
```

KFLAG

Function

Sets kernel Flags

Extension Group

BASELINE

Operation

if (cc) assign bits from operand to STATUS32

Instruction Format

op c

Syntax Example

KFLAG<.cc> c

STATUS32 Flags Affected

Flag		Mode	Source of operand
IE	•	Kernel	Bit 31 of Source Operand
US	•	Kernel	Bit 20 of Source Operand
AD	•	Kernel	Bit 19 of Source Operand
RB[2:0]	•	Kernel	Bits [18:16] of Source Operand, if RGF_NUM_BANKS > 1. If RGF_NUM_BANKS == 1, RB[2:0] is read as zero and ignored on write.
SC	•	Kernel	Bit 14 of Source Operand if stack checking is configured
DZ	•	Kernel	Bit 13 of Source Operand, if DIV_REM_OPTION is configured
L			Unchanged
Z	•	Any	Bit 11 of Source Operand
N	•	Any	Bit 10 of Source Operand
C	•	Any	Bit 9 of Source Operand
V	•	Any	Bit 8 of Source Operand
U			Unchanged
DE			Unchanged

AE	•	Kernel	Bit 5 of Source Operand
E[3:0]	•	Kernel	Bit [4:1] of Source Operand
H	•	Kernel	Bit 0 of Source Operand (If set, ignore all other flags states)
H	•	Secure Kernel	Bit 0 of Source Operand (If set, ignore all other flags states)

Description

The contents of the source operand (c) are used to set the condition code and processor control flags held in the processor status registers.

Each bit of the STATUS32 register is assigned the corresponding bit from the 32-bit source operand c, according to the list in [STATUS32 Flags Affected](#). The format of the source operand is identical to the format used by the STATUS32 register (auxiliary address 0x0A).

If the H flag is set (halt processor flag), all other flag states are ignored and are not updated. If you attempt to set STATUS32.H to 1 in the normal kernel mode, a Privilege Violation is generated. Attempts to set the H bit using KFLAG in both normal user mode and secure user mode are ignored.

The flag setting field, F, is always encoded as 1 for this instruction.

The KFLAG instruction is serializing. This serialization ensures that no further instructions can be completed before all flag updates take effect.

The KFLAG instruction can be executed in any secure/normal kernel mode or user mode. However, in the user mode, KFLAG behaves as a FLAG instruction and only modifies bits Z, N, C, and V.



Note

Using the KFLAG instruction to place the processor in Exception or Interrupt mode does not alter the zero-overhead loop disable bit, L. The L bit is set automatically only when a genuine exception or interrupt causes the processor to enter Exception or Interrupt modes, or when this bit is restored by an RTIE instruction.

Pseudo Code

```
if ((src[0] == 1) && (STATUS32[7] == 0)) then                                KFLAG
    STATUS32[0] = 1
    Halt()
else
    STATUS32[11:8] = src[11:8]
    if (STATUS32[7] == 0 && HAS_INTERRUPTS > 0) /* in kernel mode */
        STATUS32[5:1] = src[5:1]
        if (DIV_REM_OPTION == 1)
            STATUS32[13] = src[13]
        if (STACK_CHECKING == 1)
            STATUS32[14] = src[14]
        if (RGF_NUM_BANKS > 1)
            STATUS32[b:16] = src[b:16] /* b = 15+log2(RGF_NUM_BANKS) */
        if (LL64_OPTION == 1)
            STATUS32[19] = src[19]
    STATUS32[20] = src[20]
    STATUS32[31] = src[31]
```

<pre> if ((src[0] == 1) && (STATUS32.U == 0)) then if (STATUS32.S == 1) then STATUS32.H = 1 Halt() else /* in non-secure mode */ exception(EV_PrivilegeV, ECR = 0x071004) else STATUS32[11:8] = src[11:8] /*ZNCV, can be updated in any mode*/ if (STATUS32.U == 0) /*in kernel mode*/ if (SEC_MODES_OPTION == 1) /*SecureShield is present*/ if (STATUS32.S == 1) /*in secure mode*/ STATUS32.E = src[4:1] /*E[3:0]*/ else /*SecureShield is not present*/ STATUS32.E = src[4:1] /*E[3:0]*/ if (DIV_REM_OPTION == 1) STATUS32.DZ = src[13] /*DZ*/ if (STACK_CHECKING == 1) STATUS32.SC = src[14] /*SC*/ if (SEC_MODES_OPTION == 0) /*SecureShield is not present*/ if (RGF_NUM_BANKS > 1) STATUS32.RB = src[b:16] /* b = 15+log2(RGF_NUM_BANKS) if (LL64_OPTION == 1) STATUS32.AD = src[19] /*AD*/ STATUS32.US = src[20] /*US*/ if (SEC_MODES_OPTION == 1) /*SecureShield is present*/ if (STATUS32.S == 1) /*in secure mode*/ STATUS32.IE = src[31] /* IE */ else /*SecureShield is not present*/ STATUS32.IE = src[31] /* IE */ </pre>	<pre> KFLAG when SEC_MODE _OPTIO== 1 </pre>

Assembly Code Example

KFLAG 0x01	; Halt processor (other flags not updated)
KFLAG 0x10	; Set interrupt priority threshold to 8.

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
KFLAG	c	00100RRR001010011RRRCCCCRRRRRR
KFLAG	u6	00100RRR011010011RRRuuuuuuRRRRRR
KFLAG	s12	00100RRR101010011RRRssssssSSSSSS
KFLAG<.cc>	c	00100RRR111010011RRRCCCC0QQQQQ
KFLAG<.cc>	u6	00100RRR111010011RRRuuuuuu1QQQQQ

LD LDH LDW LDB

Function

Load from Memory



Note

The LDW mnemonic is deprecated, and equivalent to LDH

Extension Group

LD, LDH, LDB: BASELINE

LD_S R0-3,[h,u5]: CODE_DENSITY

LD_S.AS a,[b,c]: CODE_DENSITY

LD_S R1,[GP, s11]: CODE_DENSITY

Operation

addr=b+c; a = *addr;

Instruction Format

op a, b, c

Syntax Example

LD<zz><.x><.aa><.di> a,[b,c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

A memory load occurs from the address that is calculated by adding source operand 1 (b) with source operand 2 (c) and the returning load data is written into the destination register (a).

The status flags are not updated by this instruction.

**Note**

For 16-bit encoded instructions that work on the stack pointer (SP) or global pointer (GP), the offset is aligned to 32-bit. For example, LD_S b,[sp,u7] only needs to encode the top 5 bits because the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

The size of the requested data is specified by the data size field `<.zz>` and by default, data is zero extended from the most-significant bit of the data to the most-significant bit of the destination register.

Table 7-13 on page 312 lists the data size modes.

Data can be sign-extended by enabling sign extend `<.x>`. Table 7-14 on page 312 lists the sign extend modes. The `<.x>` bit is not supported by the instruction formats for the LDD and LDL instructions. These instructions never perform any sign extension of the values they load from memory.

If the processor contains a data cache, load requests can bypass the cache by using the `<.di>` syntax. Table 7-12 on page 311 lists the direct to memory bypass modes.

The address write-back mode can be selected by using the `<.aa>` syntax. When using the scaled source addressing mode (`.AS`), the scale factor is dependent upon the size of the data word requested (`.zz`).

Table 7-10 on page 310 lists the address write-back modes.

**Note**

LP_COUNT may not be used as the destination of a load. For example, the following instruction is *not* allowed: LD LP_COUNT, [r0].

Using the sign-extend suffix on the LD instruction with a 32-bit data size raises an **Illegal Instruction** exception.

A load that specifies a null destination register, effectively performs a pre-fetch operation.

Using the sign extension suffix with an LDD instruction raises an **Illegal Instruction** exception.

Pseudo Code

```
if AA==0 then address = src1 + src2    /* LD */
if AA==1 then address = src1 + src2
if AA==2 then address = src1
if AA==3 and ZZ==0 then
    address = src1 + (src2 << 2)
if AA==3 and ZZ==2 then
    address = src1 + (src2 << 1)
if AA==3 and ZZ==3 then
    address = src1 + (src2 << 2)
if AA==1 or AA==2 then
    src1 = src1 + src2
DEBUG[LD] = 1
```

```

dest = Memory(address, size)          /* On Returning Load */
if X==1 then
    dest = Sign_Extend(dest, size)
if NoFurtherLoadsPending() then
    DEBUG[LD] = 0

```

Assembly Code Example

```
LD r0, [r1,4]    ; Load word from memory address r1+4 and write result to r0
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
LD<zz><.x><.aa><.di>	a, [b]	00010bbb000000000BBBDaaZZXAAAAAA
LD<zz><.x><.aa><.di>	a, [b, s9]	00010bbbssssssssSBBBDaaZZXAAAAAA
LD<zz><.x><.aa><.di>	a, [b, c]	00100bbbaa110ZZXD BBBCCCCC AAAAAA
LD_S	a, [b, c]	01100bbbccc00aaa
LD_S	b, [SP, u7]	11000bbb000uuuuu
LD_S	b, [PCL, u10]	11010bbbuuuuuuuu
LD_S	c, [b, u7]	10000bbbcccuuuuu
LD_S	R0, [GP, s11]	1100100sssssssss
LDB_S	a, [b, c]	01100bbbccc01aaa
LDB_S	b, [SP, u7]	11000bbb001uuuuu
LDB_S	c, [b, u5]	10001bbbcccuuuuu
LDB_S	R0, [GP, s9]	1100101sssssssss
LDH_S	a, [b, c]	01100bbbccc10aaa
LDH_S	c, [b, u6]	10010bbbcccuuuuu
LDH_S	R0, [GP, s10]	1100110sssssssss
LDH_S.X	c, [b, u6]	10011bbbcccuuuuu

**Note**

The encoding of `LDB_S b, [SP,u7]` includes a 5-bit constant, but this constant gets shifted left by two bit positions, as all stack addresses are expected to be four-byte-aligned. This alignment makes the constant a u7 when considering the range of offsets, although the lower two bits are always 0.

Encodings without destination register

<code>LD<zz><.x><.aa><.di></code>	<code>0, [b,s9]</code>	<code>00010bbbssssssssSBBBDaaZZX111110</code>
<code>LD<zz><.x><.aa><.di></code>	<code>0, [b,c]</code>	<code>00100bbbbaa110ZZXD BBBCCCCC111110</code>

Encodings supported by the CODE DENSITY options

<code>LD_S</code>	<code>R0, [h,u5]</code>	<code>01000U00hhhhuu1HH</code>
<code>LD_S</code>	<code>R1, [h,u5]</code>	<code>01000U01hhhhuu1HH</code>
<code>LD_S</code>	<code>R2, [h,u5]</code>	<code>01000U10hhhhuu1HH</code>
<code>LD_S</code>	<code>R3, [h,u5]</code>	<code>01000U11hhhhuu1HH</code>
<code>LD_S.AS</code>	<code>a, [b,c]</code>	<code>01001bbbccc00aaa</code>
<code>LD_S</code>	<code>R1, [GP,s11]</code>	<code>01010SSSSSS00sss</code>

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

LDI, LDI_S

Function

Load Indexed

Extension Group

CODE_DENSITY

Operation

$\text{addr} = \text{AUX\_LDI\_BASE} + (\text{src2} \ll 2); b = * \text{addr};$

Instruction Format

op b, [u7]

op b, [src2]

Syntax Example

LDI_S b, [u7]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

A memory address is computed as the sum of the AUX_LDI_BASE register and the scaled unsigned index operand, and the 32-bit word at that location in memory is read. The value read from memory is placed in the destination b register.

Pseudo Code

```
b = mem [AUX_LDI_BASE + (u7 << 2)]      /* LDI_S */
b = mem [AUX_LDI_BASE + (src2 << 2)]     /* LDI   */
```

Assembly Code Example

```
LDI_S r0, [45]    ; load into r0 from the location given by
                  ;AUX_LDI_BASE + 180
```


Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LDI	b, [c]	00100bbb00100110RBBBCCCCCRRRRRR
LDI	b, [u6]	00100bbb01100110RBBBuuuuuuRRRRRR
LDI	b, [s12]	00100bbb10100110RBBBssssssSSSSSS
LDI_S	b, u7	01010bbbUUUU1uuu

LEAVE_S

Function

Function epilog sequence

Extension Group

CODE_DENSITY

Operation

Pop a collection of registers from the stack, updating stack pointer and frame pointer, as required, and optionally jumping to the return address given by the contents of r31 (blink).

Instruction Format

op u7

Syntax Example

LEAVE_S {r13-r26, fp, blink, pcl}

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

This instruction performs the function epilog code, loading from the stack the set of registers defined by the 7-bit literal operand. On completion of this instruction, the stack pointer (r28) is incremented by 4S, where S is the total number of registers loaded.

For ENTER_S and LEAVE_S instructions, if the stack pointer is not 32-bit aligned, a misaligned access exception is always raised irrespective of the STATUS32.AD bit.

This instruction can optionally jump to the location given by the contents of the blink register, after all register loads are completed.

The status flags are not updated by this instruction.

The LEAVE_S instruction is not allowed to appear in the delay slot of a jump or branch instruction. The processor raises an Illegal Instruction Sequence exception on any attempt to execute LEAVE_S in a delay slot.

**Note**

An illegal instruction sequence exception is not raised when the LEAVE_S instruction has zero operands.

The u7 encoding is as follows:

- u[3:0] Indicates the number of general-purpose registers to be restored from the stack, starting from r13 and counting contiguously upwards. When 32 general-purpose registers are configured, at most 14 registers (r13 to r26) can be restored. When only 16 general-purpose registers are configured, at most 3 registers (r13 to r15) can be restored. Therefore, if u[3:0] > 14 in a 32-register configuration, or if u[3:0] > 3 in a 16-register configuration, an Illegal Instruction exception is raised.
- u[4] If set to 1, r28 (stack-pointer) is set to r27 (frame-pointer) before registers are restored from the stack.
The frame-pointer is also restored from the stack.
- u[5] If set to 1, the blink register (r31) is restored.
- u[6] If set to 1, jump to the location given by the contents of the blink registers (r31) on completion of all register loads.

This instruction may perform as many as 16 reads from memory. These behave identically to a normal load instruction, such as:

```
LD r31, [sp,+16]
```

Pseudo Code

```

if (RGF_NUM_REGS == 32)
    MaxRegs = 14
else
    MaxRegs = 3

if (u[3:0] > MaxRegs)
    RaiseException (IllegalInstruction)

if (InDelaySlot)
    RaiseException (IllegalInstructionSequence)

M = 0                                // initial sp offset = 0
W = 4

if (u[4] == 1)                       // if frame-pointer saved
    sp = r[27]                       // then set sp = fp

if (u[5] == 1)                       // restore blink?
{
    r31 = mem[sp + M]
    M += W
}

for (i = 0; i < u[3:0]; i++)
{
    r[i+13] = mem[sp + M]           // restore GPR i+13
    M += W
}

if (u[4] == 1)                       // restore frame pointer?
{
    r27 = mem[sp + M]
    M += W
}

sp = sp + M                          // deallocate stack space

if (u[6] == 1)                       // if return jump required
    PC = r31                        // jump [blink]

```

Assembly Code Example

```

LEAVE_S {r13-r26, blink, pcl}      ;Pop registers r13-r26, and blink
                                   ;from the stack, updating stack
                                   ;pointer.
                                   ;Afterwards, jump to the return
                                   ;address restored to blink.

```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code

LEAVE_S u7 11000UUU110uuuu0

LP<cc>

Function

Loop Set Up

Extension Group

BASELINE

Operation

if (!cc) PC = PCL + u7 else {LP_END = PCL + u7; LP_START = NEXT_PC;}

Instruction Format

op u7

op s13

Syntax Example

LP<cc> u7

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

This instruction sets up a zero-overhead loop that may be conditional or unconditional, and has a single literal operand which specifies the displacement to the end of the loop.

The literal operand (u7 or s13) specifies a relative displacement value that is added to the current 32-bit word-aligned PC value (PCL) to determine the loop-end address. This address represents the address of the first instruction after the zero-overhead loop body.

The number of iterations through the loop is nominally set by the LP_COUNT register. If LP_COUNT is set to 1, the loop body executes once. The LP_COUNT value is tested at the end of the loop for the loop termination value of 0x1, decremented before the next iteration. If the LP_COUNT register value is zero when the loop begins, the value at the end of the 1st iteration is decremented to 0xFFFF_FFFF (wraps), and the loop body is executed $(2^{32} - 1)$ times. Loops may be exited early by branch or jump instructions within the loop.

If a condition is specified by the LPcc instruction and the condition is *not* met, the program execution is resumed from the 16-bit aligned loop-end address.

If the condition is met, or if the instruction does not specify a condition, the auxiliary register LP_END (auxiliary register 0x03) is assigned the loop-end address. In parallel, the LP_START register (auxiliary register 0x02) is assigned the address of the next instruction after LPcc. These assignments take effect on completion of the LPcc instruction.

The non-conditional LP instruction always updates LP_END and LP_START auxiliary registers.

The conditional execution is summarized in the following table:

Loop Format	LPcc Semantics (Conditional Execution <cc>)	
	True	False
LPcc u7	$\text{aux\_reg}[\text{LP\_END}] = \text{PCL} + \text{u7}$ $\text{aux\_reg}[\text{LP\_START}] = \text{NEXT\_PC}$	$\text{PC} = \text{PCL} + \text{u7}$
LP s13	$\text{aux\_reg}[\text{LP\_END}] = \text{PCL} + \text{s13}$ $\text{aux\_reg}[\text{LP\_START}] = \text{NEXT\_PC}$	Not possible

Table 7-7 on page 308 lists the condition codes, cc.



Caution

The LPcc instruction must not be in the executed delay slot of branch or jump instructions, and therefore, the LPcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBitn.D instruction.

Because LP_END is set to 0 upon [Reset](#), you must not execute an instruction placed at the end of program memory space (0xFFFFF0FC or 0xFFFFF0FE) because such instructions trigger the LP mechanism if no other LP has been set up since [Reset](#).

Use caution if code is copied or overlaid into memory or if using any form of MMU or memory mapping; Ensure before executing the code that LP_END is initialized to a safe value (0) to prevent accidental LP triggering.

Care should be taken to align LP_END with an instruction boundary position. An Illegal Instruction Sequence exception is raised when LP_END is at 16-bit offset position immediately after an RTIE (32-bit) instruction.

The LP instruction is encoded to use immediate values (syntax u7 or syntax s13). Encoding the operand mode (bits 23:22) to be 0x0 is illegal and raises an Illegal Instruction exception. In addition, using operand mode 0x3 with sub-operand mode 0x0 is illegal, and also raises an Illegal Instruction exception. The processor ignores the reserved field, R.

The LP instruction may be used to set up a loop with a maximum loop size determined by the limit of the branch offset available in each of the two LP encodings.

- Conditional LPcc – 6 bits of unsigned offset
- Unconditional LP – 12 bits of signed offset

In common with all relative branch offsets, the calculation of the LP branch target, which defines the first instruction after the loop, is given by the sum of the current 32-bit word-aligned PC (cPCL) plus

the offset. The offset is encoded as a half-word displacement, and is therefore scaled by a 1-bit left shift before being added to cPCL.

A 6-bit unsigned offset therefore provides a maximum displacement of 126 bytes from a 32-bit word-aligned LP instruction, or 124 bytes from an LP instruction at aligned to an odd half-word address. The LPcc instruction itself occupies the first 4 bytes of this displacement, and therefore the maximum loop size for a conditional LPcc instruction is 122 or 120 bytes for 32-bit aligned and 16-bit aligned LPcc instructions respectively.

The unconditional form of the LP instruction has a signed 12-bit offset, allowing an effective displacement range of -4096 to +4094 bytes for a 32-bit aligned LP, and -4098 to +4092 bytes for a 16-bit aligned LP instruction.

Jumps and control-transfer instructions, and their delay-slots where present, may not appear as the last instruction of a zero-overhead loop body.

One zero-overhead loop may be used inside another if the inner loop saves and restores the context of the outer loop and complies with all other rules. An additional rule is that a loop instruction may not appear in the last position, or the penultimate position, of a zero-overhead loop body.

The following instructions are prohibited in the last position (the instruction immediately before LOOP_END) of a zero-overhead loop body:

- ❑ A branch or jump instruction (with or without a delay slot and either taken or not taken) (B, Bcc, Bcc_S, BR, BRcc, BRcc_S, BBIT0, BBIT1, BL, BLcc, BL_S, BI, BIH, J, Jcc, J_S, Jcc_s, J blink, JLcc, JL_S)
- ❑ The delay-slot instruction of a taken branch or jump.



Note

If a branch is not taken it is not always possible to know that the delay-slot instruction is actually a delay slot. That is the case if there is an intervening exception.

- ❑ ENTER_S, LEAVE_S, RTIE
- ❑ EI_S
- ❑ E-slot (the execute slot of an EI_S instruction).
- ❑ JLI_S
- ❑ LPcc

If the prohibited instructions are used in the last position of a zero-overhead loop, the effect is implementation dependent. For further details, see the Databook of each ARCV2-based processor.

The use of zero delay loops is illustrated in the following example.

Example 10-3 Example Loop Code

```

MOV LP_COUNT,2      ; do loop 2 times (flags not set)
...                ; Some intermediate instructions
LP loop_end         ; set up loop mechanism to work
                    ; between loop_in and loop_end
loop_in: LR r0,[r1]  ; first instruction in loop
ADD r2,r2,r0        ; sum r0 with r2
BIC r1,r1,4         ; last instruction
                    ; in loop
loop_end:
ADD r19,r19,r20     ; first instruction after loop

```

Direct writes to the LP_START and LP_END registers can be used to set up larger loops, if required.

**Caution**

When directly manipulating LP_START and LP_END, ensure that the values written refer to the first address occupied by an instruction.

Loop Mechanism

The loop mechanism is inactive whenever the loop-inhibit bit STATUS32[L] is set to one. This bit is set, thereby disabling the loop mechanism on an interrupt or an exception (including TRAP instructions). Loops are enabled (STATUS32[L]=0) after [Reset](#). The loop-inhibit bit is cleared (loops allowed) whenever the processor commits a taken conditional LPcc instruction or an unconditional LP instruction. From kernel mode, the value of the bit can also be restored using the RTIE instruction.

When the loop mechanism is disabled (STATUS32[L]=1), loop-end conditions are ignored, no change of program flow takes place, and the LP_COUNT register is not decremented. The STATUS32[L] register does not affect the reads and writes to the loop control registers.

The processor checks for a loop-end condition when calculating the next program counter address, before each instruction is completed.

A loop-end condition is an implicit branch to the start of the loop (LP_START). A loop-end condition is detected when **all** of the following conditions are met:

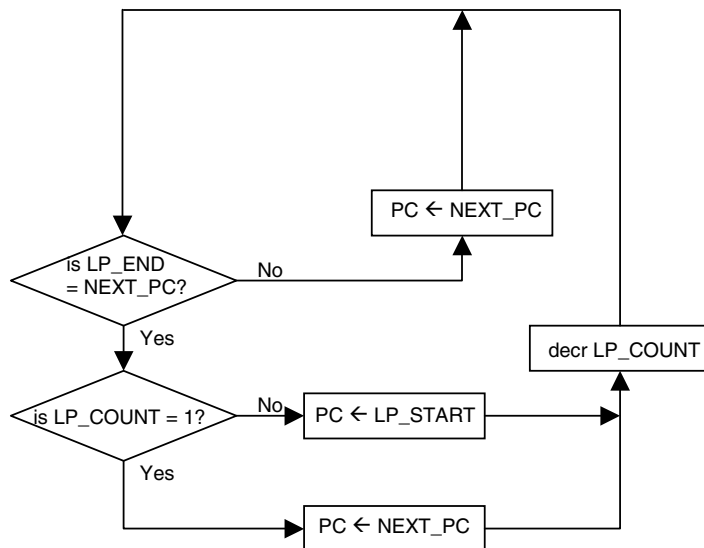
- ❑ STATUS32[DE] is 0, indicating that the instruction is not in the delay slot of a branch, or the branch was not taken.
- ❑ STATUS32[L] is 0
 - This bit is set to 1 to disable loop-end detection.
- ❑ The instruction to be completed is the last in a loop
 - $PC + \text{current instruction size} = LP\_END$
- ❑ LP_COUNT is not equal to 1
 - When LP_COUNT=1, LP_COUNT is decremented and execution continues from the instruction pointed to by LP_END.
- ❑ $(LP\_END - LP\_START) \geq 4$

- If this condition is not true, LP_START and LP_END define a loop body containing fewer than 4 bytes. This size is smaller than the minimum supported zero-overhead loop size in the ARCV2 architecture.

When a loop-end condition is detected, the processor jumps to the address in LP_START, and LP_COUNT is decremented.

If LP_COUNT is 1, the processor continues execution from the instruction pointed to by LP_END; LP_COUNT is also decremented. This sequence is illustrated in [Figure 10-1](#).

Figure 10-1 Loop Detection and Update Mechanism



The ARCV2-based processor does not prevent the LP instruction from setting up a loop of less than the minimum 4 byte size. However, in this case, the loop-end condition may not be detected, according to the rules defined earlier in this section.

If a LP_START value is provided which does not match the start of an instruction, and the loop-end condition is reached, the result is the same as if a branch or jump had been made to the faulty address.

If an LP_END value is provided which does not match the start of an instruction, the loop-end condition is never detected.

The effect of the loop mechanism when reading or writing LP_COUNT, LP_START, and LP_END in the last position of the loop is summarized in [Table 10-3](#). This table covers loop setup and use of long immediate data.



Instruction numbers islotn refer to the sequence of executed instructions within a loop, which is not the same as the static instruction positions in the code, if branches are used within the loop.

Instruction numbers oslotn refer to instructions outside the loop.

Table 10-3 When Reads and Writes Take Effect

	Instruction	Writing Takes Effect	Reading Uses	Writing Takes Effect	Reading Uses
Loop_start:		LP_COUNT	LP_COUNT	LP_END, LP_START	LP_END, LP_START
	islot1	current iteration	current iteration	current iteration	current iteration
	islot2	current iteration	current iteration	current iteration	current iteration
	islotn-3	current iteration	current iteration	current iteration	current iteration
	islotn-2	current iteration	current iteration	current iteration	current iteration
	islotn-1	current iteration	current iteration	current iteration	current iteration
	islotn	next cycle	current iteration	next cycle	current iteration
Loop_end:					
	oslot1				
	oslot2				

Pseudo Code

```

if cc==true then                                /* LPcc */
  Aux_reg(LP_START) = nPC
  Aux_reg(LP_END) = cPCL + rd
  PC = nPC
else
  PC = cPCL + rd

```

Assembly Code Example

```

LPNE label                                     ; if the Z flag is set then
                                                ; branch to label else
                                                ; set LP_START to address of
                                                ; next instruction and set
                                                ; LP_END to label

```

The use of zero delay loops is illustrated in [Example 10-4](#).

Example 10-4 Use of Zero Overhead Loop

```

        MOV LP_COUNT,2    ; do loop 2 times (flags not set)
        LP loop_end      ; set up loop mechanism to work
                           ; between loop_in and loop_end
loop_in: LR r0,[r1]       ; first instruction of loop
        ADD r2,r2,r0      ; second instruction of loop
        BIC r1,r1,4       ; last instruction of loop
loop_end:
        ADD r19,r19,r20   ; first instruction after loop

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LP<cc>	u7	00100RRR11101000RRRRuuuuuu1QQQQQ
LP	u7	00100RRR01101000RRRRuuuuuuRRRRRR
LP	s13	00100RRR10101000RRRRssssssSSSSSS

LR

Function

Load from Auxiliary Register

Extension Group

BASELINE

Operation

$b = \_lr(c);$

Instruction Format

op b,c

Syntax Example

LR b,[c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Get the data from the auxiliary register whose number is obtained from the source operand (c) and place the data into the destination register (b).

The LR instruction executes unconditionally. Therefore, specifying an operand mode (bits 23:22) of 0x3 raises an [Illegal Instruction](#).

Pseudo Code

```
dest = Aux_reg(src)          /* LR */
```

Assembly Code Example

```
LR r1, [r2]      ; Load contents of Aux. register ;pointed
                  ; to by r2 into r1
```

Syntax and Encoding

The status flags are not updated with this instruction therefore the flag setting field, F, is encoded as 0. The reserved field, R, is ignored by the processor, but must be set to 0.

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LR	b, [c]	00100bbb00101010RBBBCCCCCRRRRRR
LR	b, [u6]	00100bbb01101010RBBBuuuuuuRRRRRR
LR	b, [s12]	00100bbb10101010RBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

LSL16

Function

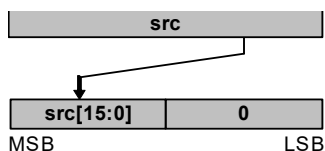
Logical Shift Left 16

Extension Group

SWAP_OPTION

Operation

$b = c \ll 16;$



Instruction Format

op b,c

Syntax Example

LSL16<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 16 places to the left, clearing the lower 16 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src << 16                                     /* LSL16 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
LSL16 r1,r2           ; Logical shift of r2 16
                      ;places to the
                      ; left, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

LSL16<.f>	b, c	00101	bbb00	101111	FBBB	CCCCC	001010
LSL16<.f>	b, u6	00101	bbb01	101111	FBBB	uuuuuu	001010

LSL8

Function

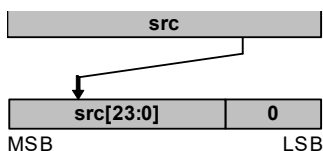
Logical Shift Left 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$b = c \ll 8;$



Instruction Format

op b,c

Syntax Example

LSL8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 8 places to the left, clearing the lower 8 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src << 8                                     /* LSL8 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
LSR8 r1,r2      ; Logical shift of r2 8 places to the left, placing
                ;result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

LSL8<.f>	b,c	00101	bbb00	101111	FBBB	CCCCC	001111
LSL8<.f>	b,u6	00101	bbb01	101111	FBBB	uuuuuu	001111

LSR

Function

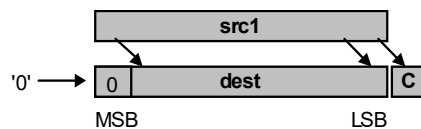
Logical Shift Right

Extension Group

BASELINE

Operation

$b = (\text{unsigned})\ c \gg 1;$



Instruction Format

op b,c

Syntax Example

LSR<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Logically right shift the source operand (c) by one and place the result into the destination register (b).

The most-significant bit of the result is replaced with 0.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 1                                /* LSR */
dest[31] = 0
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[0]
```

Assembly Code Example

```
LSR r1,r2    ; Logical shift right contents of r2 by one bit and write
              ;result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LSR<.f>	b,c	00100bbb00101111FBBBCCCCC000010
LSR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000010
LSR_S	b,c	01111bbbccc11101

LSR multiple

Function

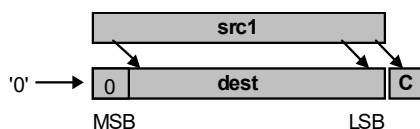
Multiple Logical Shift Right

Extension Group

SHIFT_OPTION 2 or 3

Operation

if (cc) a = (unsigned) b >> c;



Instruction Format

op a, b, c

Syntax Example

LSR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Logically shift right b by c places and place the result in the destination register. Only the bottom 5 bits of c are used as the shift value.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* LSR
dest = src1 >> (src2 & 31)                      Multiple */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = if src2==0 then 0 else src1[src2-1]

```

Assembly Code Example

```

LSR r1,r2,r3    ; Logical shift right contents of r2 by r3 bits and
                ;write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LSR<.f>	a,b,c	00101bbb0000001FBBBCCCCCAAAAAA
LSR<.f>	a,b,u6	00101bbb01000001FBBBuuuuuuAAAAAA
LSR<.f>	b,b,s12	00101bbb1000001FBBBssssssSSSSSS
LSR<.cc><.f>	b,b,c	00101bbb11000001FBBBCCCCC0QQQQQ
LSR<.cc><.f>	b,b,u6	00101bbb11000001FBBBuuuuuu1QQQQQ
LSR_S	b,b,c	01111bbbccc11001
LSR_S	b,b,u5	10111bbb001uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

LSR16

Function

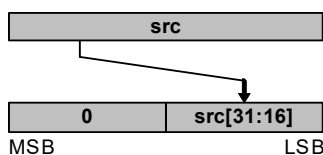
Logical Shift Right 16

Extension Group

SWAP_OPTION

Operation

$b = c \gg 16;$



Instruction Format

op b,c

Syntax Example

LSR16<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= 0
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 16 places to the right, clearing the upper 16 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 16                                /* LSR16 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```


Assembly Code Example

```
LSR16 r1,r2    ; Logical shift of r2 16 places to the right, placing
               ;result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

LSR16<.f>	b, c	00101	bbb00	1011111	FBBB	CCCCC	001011
LSR16<.f>	b, u6	00101	bbb01	1011111	FBBB	uuuuuu	001011

LSR8

Function

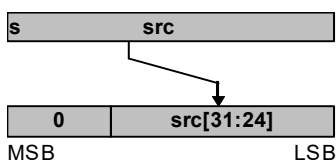
Logical Shift Right 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$b = c \gg 8;$



Instruction Format

op b,c

Syntax Example

LSR8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= 0
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 8 places to the right, clearing the upper 8 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 8                                     /* LSR8 */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
LSR8 r1,r2    ; Logical shift of r2 8 places to the
               ; right, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

LSR8<.f>	b,c	00101	bbb00	101111	FBBB	CCCCC	001110
LSR8<.f>	b,u6	00101	bbb01	101111	FBBB	uuuuuu	001110

MAX

Function

Return Maximum Value

Extension Group

BASELINE

Operation

if (cc) a = (b>c) ? b : c;

Instruction Format

op a, b, c

Syntax Example

MAX<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if both source operands are equal
N	•	= Set if most-significant bit of result of src1-src2 is set
C	•	= Set if src2 is selected (src2 >= src1)
V	•	= Set if an overflow is generated (as a result of src1-src2)

Description

Return the maximum of the two signed source operands (b and c) and place the result in the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* MAX */
    alu = src1 - src2
    if src2 >= src1 then
        dest = src2
    else
        dest = src1
    if F==1 then
        Z_flag = if alu==0 then 1 else 0
        N_flag = alu[31]
        V_flag = Overflow()
        C_flag = if src2>=src1 then 1 else 0

```

Assembly Code Example

```
MAX r1,r2,r3    ; Take maximum of r2 and r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MAX<.f>	a,b,c	00100bbb00001000FBBBCCCCC	AAAAAA
MAX<.f>	a,b,u6	00100bbb01001000FBBBuuuuuu	AAAAAA
MAX<.f>	b,b,s12	00100bbb10001000FBBBssssss	SSSSSS
MAX<.cc><.f>	b,b,c	00100bbb11001000FBBBCCCCC	0QQQQQ
MAX<.cc><.f>	b,b,u6	00100bbb11001000FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MIN

Function

Return Minimum Value

Extension Group

BASELINE

Operation

if (cc) a = (b < c) ? b : c;

Instruction Format

op a, b, c

Syntax Example

MIN<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if both source operands are equal
N	•	= Set if most-significant bit of result of src1-src2 is set
C	•	= Set if src2 is selected (src2 <= src1)
V	•	= Set if an overflow is generated (as a result of src1-src2)

Description

Return the minimum of the two signed source operands (b and c) and place the result in the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* MIN */
    alu = src1 - src2
    if src2 <= src1 then
        dest = src2
    else
        dest = src1
    if F==1 then
        Z_flag = if alu==0 then 1 else 0
        N_flag = alu[31]
        V_flag = Overflow()
        C_flag = if src2<=src1 then 1 else 0

```

Assembly Code Example

```
MIN r1,r2,r3    ; Take minimum of r2 and r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MIN<.f>	a,b,c	00100bbb00001001FBBBCCCCC	AAAAAA
MIN<.f>	a,b,u6	00100bbb01001001FBBBuuuuuu	AAAAAA
MIN<.f>	b,b,s12	00100bbb10001001FBBBssssss	SSSSSS
MIN<.cc><.f>	b,b,c	00100bbb11001001FBBBCCCCC	0QQQQQ
MIN<.cc><.f>	b,b,u6	00100bbb11001001FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MOV

Function

Move Contents

Extension Group

BASELINE

Operation

if (cc) b = c;

Instruction Format

op b, c

Syntax Example

MOV<.cc><.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

The contents of the source operand (c) are moved to the destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then /* MOV */           /* MOV */
  dest = src
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
MOV r1,r2      ; Move contents of r2 into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
MOV<.f>	b,c	00100bbb00001010FBBBCCCCCRRRRRR
MOV<.f>	b,u6	00100bbb01001010FBBBuuuuuuRRRRRR
MOV<.f>	b,s12	00100bbb10001010FBBBssssssSSSSSS
MOV<.cc><.f>	b,c	00100bbb11001010FBBBCCCCC0QQQQQ
MOV<.cc><.f>	b,u6	00100bbb11001010FBBBuuuuuu1QQQQQ
MOV_S	h,s3	01110ssshhh011HH
MOV_S	0,s3	01110sss11001111
MOV_S.NE	b,h	01110bbbh hh111HH
MOV_S.NE	b,limm	01110bbb11011111
MOV_S	b,u8	11011bbbuuuuuuuu
MOV_S	g,h	01000ggghhhGG0HH
MOV_S	g,limm	01000ggg110GG011
MOV_S	0,h	01000110hhh110HH
MOV_S	0,limm	0100011011011011

MPY, MPY_S

Function

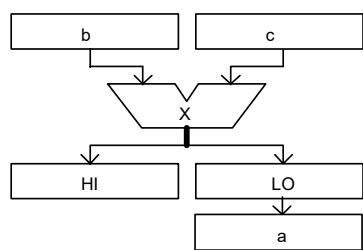
32 x 32 Signed Multiply, returning least-significant 32-bit word of the result. For the DSP version, see [MPY](#).

Extension Group

(MPY_OPTION > 1) or (HAS_DSP == 1)

Operation

if (cc) a = (signed) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPY<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 64-bit result is set
C		= Unchanged
V	•	= Set when the signed result cannot be wholly contained within the lower part of the 64-bit result. In other words, when bits 62:31 do not equal bit 64, the sign bit.

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination operand register.

Pseudo Code

```
if cc==true then                                     /* MPY */
    dest = (src1 * src2) & 0x0000_0000_FFFF_FFFF
```

Assembly Code Example

```
MPY r1,r2,r3          ; Multiply r2 by r3 and put the least
                      ; significant word of the result in r1
MPY_S r1,r2           ; 16-bit encoding of MPY r1,r1,r2
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
MPY<.f>	a,b,c	00100bbb00011010FBBBCCCCC0AAAAAA
MPY<.f>	a,b,u6	00100bbb01011010FBBBuuuuuuAAAAAA
MPY<.f>	b,b,s12	00100bbb10011010FBBBssssssSSSSSS
MPY<.cc><.f>	b,b,c	00100bbb11011010FBBBCCCCC0QQQQQ
MPY<.cc><.f>	b,b,u6	00100bbb11011010FBBBuuuuuu1QQQQQ
MPY_S	b,b,c	01111bbbccc01100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYM MPYH

Function

32 x 32 Signed Multiply High. For the DSP version, see [MPYM MPYH](#)

 **Note**

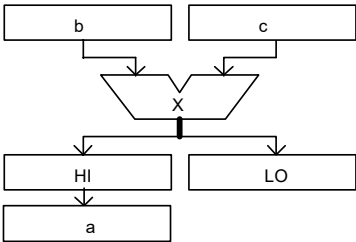
The MPYH mnemonic is deprecated.

Extension Group

(MPY_OPTION > 1) or (HAS_DSP == 1)

Operation

if (cc) a = (signed) (b * c) >> 32;



Instruction Format

op a, b, c

Syntax Example

MPYM<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set is the result is negative
C		= Unchanged
V	•	= Always cleared.

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2, placing the most-significant 32 bits of the 64-bit result in the destination register. Any flag updates occur only if the set

flags suffix (.F) is used. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if cc==true then                                     /*MPYM*/
  dest = (src1 * src2) >> 32
```

Assembly Code Example

```
MPYM r1,r2,r3      ;Multiply r2 by r3 and put high part of the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MPYM<.f>	a,b,c	00100bbb00011011FBBBCCCCCAAAAA
MPYM<.f>	a,b,u6	00100bbb01011011FBBBuuuuuuAAAAA
MPYM<.f>	b,b,s12	00100bbb10011011FBBBssssssSSSSSS
MPYM<.cc><.f>	b,b,c	00100bbb11011011FBBBCCCCC0QQQQQ
MPYM<.cc><.f>	b,b,u6	00100bbb11011011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYMU MPYHU

Function

32 x 32 Unsigned integer Multiply High. For the DSP version, see [MPYMU MPYHU](#)



Note

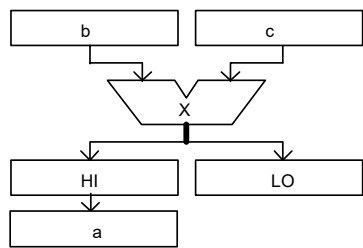
The MPYHU mnemonic is deprecated.

Extension Group

(MPY_OPTION > 1) or (HAS_DSP == 1)

Operation

if (cc) a = (unsigned) (b * c) >> 32;dest (src1 X src2).high



Instruction Format

op a, b, c

Syntax Example

MPYMU<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared.
C		= Unchanged
V	•	= Always cleared.

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand 1 and operand 2, placing the most-significant 32 bits of the 64-bit result in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used. Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if cc==true then                                /*MPYMU*/
    dest = (src1 * src2) >> 32                 /*when MPY_OPTION==2
```

Assembly Code Example

```
MPYMU r1,r2,r3    ;Multiply r2 by r3 and put high part of the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MPYMU<.f>	a,b,c	00100bbb00011100FBBBCCCCCAAAAAA
MPYMU<.f>	a,b,u6	00100bbb01011100FBBBuuuuuuAAAAAA
MPYMU<.f>	b,b,s12	00100bbb10011100FBBBssssssSSSSSS
MPYMU<.cc><.f>	b,b,c	00100bbb11011100FBBBCCCCC0QQQQQ
MPYMU<.cc><.f>	b,b,u6	00100bbb11011100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYU

Function

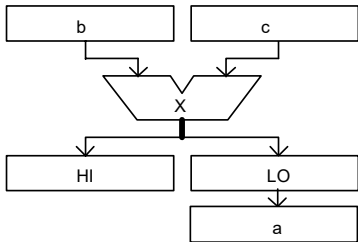
32 x 32 Unsigned Multiply, returning least-significant 32-bit word of result. For the DSP version, see [MPYU](#).

Extension Group

(MPY_OPTION > 1) or (HAS_DSP == 1)

Operation

if (cc) a = (unsigned) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPYU<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Set when the result cannot be represented as a 32-bit unsigned integer

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand1 and operand2, placing the least-significant 32 bits of the 64-bit result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if cc==true then                                     /* MPYU */
    dest = (src1 * src2) & 0x0000_0000_FFFF_FFFF
```

Assembly Code Example

```
MPYU r1,r2,r3    ;Multiply r2 by r3 and put low part of the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
MPYU<.f>	a,b,c	00100bbb00011101FBBBCCCCCAAAAA
MPYU<.f>	a,b,u6	00100bbb01011101FBBBuuuuuuAAAAA
MPYU<.f>	b,b,s12	00100bbb10011101FBBBssssssSSSSSS
MPYU<.cc><.f>	b,b,c	00100bbb11011101FBBBCCCCC0QQQQQ
MPYU<.cc><.f>	b,b,u6	00100bbb11011101FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYUW, MPYUW_S

Function

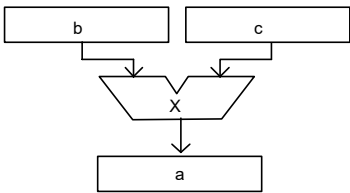
16 x 16 unsigned Multiply, return 32-bit word result. For the DSP version, see [MPYUW](#).

Extension Group

(MPY_OPTION > 0) or (HAS_DSP == 1)

Operation

if (cc) a = (unsigned) ((word) b * (word) c)



Instruction Format

op a, b, c

Syntax Example

MPYUW<.f> a,b,c
MPYUW_S b,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Always cleared

Description

Perform an unsigned 16-bit by 16-bit multiply of operand 1 and operand 2, and place 32-bit result in the destination register. The 16-bit source operands use the lower 16 bits of the source operands.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                     /*MPYUW*/
    dest = src1 * src2
```

Assembly Code Example

```
MPYUW r1,r2, r3          ; Multiply r2 by r3 and put the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MPYUW<.f>	a,b,c	00100bbb00011111FBBBCCCCCAAAAAA
MPYUW<.f>	a,b,u6	00100bbb01011111FBBBuuuuuuAAAAAA
MPYUW<.f>	b,b,s12	00100bbb10011111FBBBssssssSSSSSS
MPYUW<.cc><.f>	b,b,c	00100bbb11011111FBBBCCCCC0QQQQQ
MPYUW<.cc><.f>	b,b,u6	00100bbb11011111FBBBuuuuuu1QQQQQ
MPYUW_S	b,b,c	01111bbbccc01010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYW, MPYW_S

Function

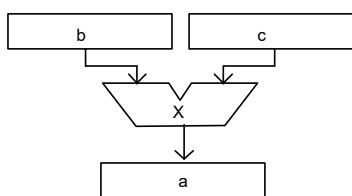
16 x 16 Signed Multiply. For the DSP version, see [MPYW](#).

Extension Group

(MPY_OPTION > 0) or (HAS_DSP == 1)

Operation

if (cc) a = (signed) ((word) b * (word) c)



Instruction Format

op a, b, c

Syntax Example

MPYW<.f> a,b,c

MPYW_S b,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 32-bit result is set
C		= Unchanged
V	•	= Set when the unsigned result overflows

Description

Perform a signed 16-bit by 16-bit multiply of operand 1 and operand 2, and place the 32-bit result in the destination register. The 16-bit source operands use the lower 16 bits of the source operands.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                /* MPYW */
    dest = src1 * src2
```

Assembly Code Example

```
MPYW r1,r2, r3          ; Multiply r2 by r3 and put the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MPYW<.f>	a,b,c	00100bbb00011110FBBBCCCCCAAAAAA
MPYW<.f>	a,b,u6	00100bbb01011110FBBBuuuuuuAAAAAA
MPYW<.f>	b,b,s12	00100bbb10011110FBBBssssssSSSSSS
MPYW<.cc><.f>	b,b,c	00100bbb11011110FBBBCCCCC0QQQQQ
MPYW<.cc><.f>	b,b,u6	00100bbb11011110FBBBuuuuuu1QQQQQ
MPYW_S	b,b,c	01111bbbccc01001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

NEG

Function

Negate

Extension Group

BASELINE

Operation

if (cc) $a = 0 - b$;

Instruction Format

op a,b

Syntax Example

NEG<.f> a,b

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

The negate instruction subtracts the source operand (b) from zero and places the result into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* NEG */
    dest = 0 - src1
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
NEG r1,r2      ; Negate r2 and write result into r1
```

Syntax and Encoding

The 32-bit instruction format is an encoding of the reverse subtract instruction, [RSUB](#), using an unsigned 6-bit immediate value set to 0.

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

NEG<.f>	a,b	00100bbb01001110FBBB000000AAAAAA
NEG<.cc><.f>	b,b	00100bbb11001110FBBB0000001QQQQQ
NEG_S	b,c	01111bbbccc10011

NOP

Function

No Operation

Extension Group

BASELINE

Operation

No Operation

Instruction Format

op

Syntax Example

NOP

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

No operation. The processor advances to the next instruction.

Pseudo Code

```
/* NOP_S */
```

Assembly Code Example

```
NOP_S ; No operation
```

Syntax and Encoding

The 32-bit NOP is an encoding of the MOV instruction (syntax MOV 0,u6) using the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 369](#). For more information about the encodings, see

[“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code

NOP_S	0111100011100000
NOP	00100110010010100111000000000000

NEXTTASK

Function

Switch to the next task and switch the register bank. Start execution from the new PC

Extension Group

Task Switching

Operation

Multi-cycle blocking

Instruction Format

op c

Syntax Example

NEXTTASK c

STATUS32 Flags Affected

Z	•	= Copied from the STATUS32 register in the shadow bank
N	•	= Copied from the STATUS32 register in the shadow bank
C	•	= Copied from the STATUS32 register in the shadow bank
V	•	= Copied from the STATUS32 register in the shadow bank

Description

The functionality of this instruction depends on the task switch controller inputs

- When this instruction is committed and `arc_nxttsk_stall` is high and `arc_nxttsk_sleep` is low, this instruction stalls the pipeline.
- When this instruction is committed and `arc_nxttsk_sleep` is high, the processor enters sleep state and wakes up only when `arc_nxttsk_sleep` goes low.
- When this instruction is retired, if `arc_nxttsk_switch` is high, it switches the active register bank and starts execution from the PC pointed by the new register bank.
- When this instruction is retired and `arc_nxttsk_switch` is low, it continues execution from next PC and does not affect the state of the core.

NEXTTASK is kernel-only instruction. You must use this instruction manually in your code. As there are only two tasks, you need not explicitly specify TASK Number. Therefore, operand to NEXTTASK instruction is ignored.

When NEXTTASK is executed from user mode, a Privilege Violation Exception is raised.

An Illegal Instruction Sequence Exception is raised, if NEXTTASK is placed:

- Inside an Exception Routine (that is, STATUS32.AE = 1)
- Inside Interrupt Sub-Routine (that is, AUX_IRQ_ACT != 0)
- In the delay slot of branch/jump instruction
- In the execution slot of an EI_S instruction
- As the last instruction in a zero overhead loop

lp_count, lp_start, and lp_end registers are not copied when the NEXTTASK commits. Therefore, if NEXTTASK instruction is used inside a zero overhead loop (except as the last instruction of zol), your software must back up the lp_count, lp_start, and lp_end registers.



Caution

If Software modifies STATUS32.AE or AUX_IRQ_ACT[15:0], then the Hardware may take an exception for NEXTTASK outside of interrupt/exception routine, or may allow NEXTTASK inside interrupt/exception routine depending on the STATUS32.AE or AUX_IRQ_ACT[15:0] value.

Pseudo Code

```
NEXTTASK r0 ; Switch to new task by switching
              the register bank and PC specified
              by new register bank.
```

Assembly Code Example

```
NEXTTASK r0 ; Switch to new task by switching the register
              bank and PC specified by new register bank.
```

Syntax and Encoding

Instruction Code

NEXTTASK	c	00100000001011110011CCCCC111111
NEXTTASK	u6	00100000011011110011uuuuuu111111

NORM

Function

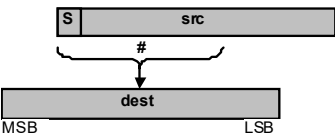
Normalize

Extension Group

BITSCAN_OPTION

Operation

```
for (i=0; (i<=31) && (c>>i != 0) && (c>>i != -1); i++); b= 31-i;
```



Instruction Format

op b,c

Syntax Example

NORM<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Computes the normalization integer for the signed value in the operand c. The normalization integer is the amount by which the operand must be shifted left to normalize the operand as a 32-bit signed integer.

Any flag updates occur only if the set flags suffix (.F) is used. The returned value for source operand of zero is 0x0000001F. Examples of returned values are shown in the following table:

Operand Value	Returned Value
0x00000000	0x0000001F
0x00000001	0x0000001E
0x1FFFFFFF	0x00000002
0x3FFFFFFF	0x00000001
0x7FFFFFFF	0x00000000
0x80000000	0x00000000
0xC0000000	0x00000001
0xE0000000	0x00000002
0xFFFFFFFF	0x0000001F

Pseudo Code

```

for (i=0;i<=31;i++)                /* NORM */
    if (src>>i ==0) break
    if (src>>i ==-1) break
end for
dest = 31-i
if F==1 then
    Z_flag = if src==0 then 1 else 0
    N_flag = src[31]

```

Assembly Code Example

```

NORM r1,r2    ; Normalization integer for r2, write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```

NORM<.f>   b,c    00101bbb00101111FBBBCCCCC000001
NORM<.f>   b,u6   00101bbb01101111FBBBuuuuuu000001

```

NORMH NORMW

Function

Normalize 16-bit Half-word



Note

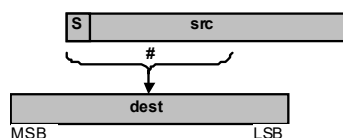
The NORMW mnemonic is deprecated.

Extension Group

BITSCAN_OPTION

Operation

```
for (i=0; (i<=15) && ((half-word) c>>i != 0) && ((half-word) c>>i != -1); i++);
b= 15-i;
```



Instruction Format

op b,c

Syntax Example

NORMH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Computes the normalization integer for the signed value in the operand. The normalization integer is the amount by which the operand must be shifted left to normalize the operand as a 16-bit signed integer. When normalizing a 16-bit signed integer, the lower 16 bits of the source data (c) is used.

Any flag updates occur only if the set flags suffix (.F) is used.

The returned value for source operand of zero is 0x000F. Examples of returned values are shown in the following table:

Operand Value	Returned Value
0x0000	0x000F
0x0001	0x000E
0x1FFF	0x0002
0x3FFF	0x0001
0x7FFF	0x0000
0x8000	0x0000
0xC000	0x0001
0xE000	0x0002
0xFFFF	0x000F

Pseudo Code

```

for (i=0;i<=15;i++)                                /* NORMW */
    if ((half-word)src>>i ==0) break
    if ((half-word)src>>i ==-1) break
end for
dest = 15-i
if F==1 then
    Z_flag = if (src & 0x0000FFFF)==0 then 1 else 0
    N_flag = src[15]

```

Assembly Code Example

```

NORMH r1,r2    ; Normalization integer for lower 16 bits of r2
               ; write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```

NORMH<.f>    b,c    00101bbb00101111FBBBCCCCC001000

```

NORMH<.f> b,u6 00101bbb01101111FBBBuuuuuu001000

NOT

Function

Bitwise NOT

Extension Group

BASELINE

Operation

$b = \sim c;$

Instruction Format

op b,c

Syntax Example

NOT<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Bitwise NOT (inversion) of the source operand (c) with the result placed into the destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = NOT(src)                                /* NOT */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
NOT r1,r2    ; Logical bitwise NOT r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

NOT<.f>	b, c	00100bbb00101111FBBBCCCCC001010
NOT<.f>	b, u6	00100bbb01101111FBBBuuuuuu001010
NOT_S	b, c	01111bbbccc10010

OR

Function

Bitwise OR

Extension Group

BASELINE

Operation

if (cc) $a = b \mid c$;

Instruction Format

op a, b, c

Syntax Example

OR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise OR of source operand 1 (b) with source operand 2 (c). The result is written into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* OR */
  dest = src1 OR src2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]

```


Assembly Code Example

```
OR r1,r2,r3    ; Logical bitwise OR contents of r2 with r3
               ; and write result into r1;
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
OR<.f>	a, b, c	00100bbb00000101FBBBCCCCCAAAAA
OR<.f>	a, b, u6	00100bbb01000101FBBBuuuuuuAAAAA
OR<.f>	b, b, s12	00100bbb10000101FBBBssssssSSSSSS
OR<.cc><.f>	b, b, c	00100bbb11000101FBBBCCCCC0QQQQQ
OR<.cc><.f>	b, b, u6	00100bbb11000101FBBBuuuuuu1QQQQQ
OR_S	b, b, c	01111bbbccc00101

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

POP

Function

Synonym for the LD instruction (see [LD LDH LDW LDB](#)).

Syntax Example

`POP %r1;` this instruction is equivalent to `LD.AB %r1, [%sp, 4]`.

POP_S

Function

Pop from Stack

Extension Group

BASELINE

Operation

$b = *SP; SP=SP+4;$

Instruction Format

op b

Syntax Example

POP_S b

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a 32-bit word memory load from the address specified in the implicit Stack Pointer, SP (r28), and place the result into the destination register (b).

On completion, the implicit stack pointer is automatically incremented by 4-bytes ($SP=SP+4$).

The status flags are not updated with this instruction.

This instruction is equivalent to a load word instruction LD when used with the following syntax:

LD.AB a, [SP,+4]

Pseudo Code

```
dest = Memory(SP, 4)    /* POP_S */
SP   = SP + 4
```

Assembly Code Example

```
POP_S r1      ; Load word from memory at address SP and write this word to
               ;r1, and then add 4 to SP
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

POP_S	b	11000bbb11000001
POP_S	BLINK	11000RRR11010001

PREFETCH

Function

Prefetch from Memory

Extension Group

BASELINE

Operation

addr=b+c; temp = *addr;

Instruction Format

op b,c

Syntax Example

PREFETCH<.aa> [b,c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The PREFETCH instruction is provided as a synonym for a particular encoding of the LD instruction.

- ❑ A memory load occurs from the address that is calculated by adding source operand 1 (b) with source operand 2 (c) and the returning load data is loaded into the data cache. The returning load is not written to any core register.
- ❑ The actual pre-fetch of data from memory takes place only if the effective address computed by this instruction would not trigger a data memory address-related exception had it been used as the effective address of a load instruction.
- ❑ This instruction does not raise data memory address-related exceptions such as: page fault, memory error, misaligned exception.
- ❑ PREFETCH and PREFETCHW are always guaranteed to execute, but the pre-fetch aspect of their behavior is no more than a hint, and is not part of the instruction semantics.
- ❑ The address update and the increment of the PC always take place. Depending on the address update mode, this instruction can be thought of as either a NOP or ADD.
- ❑ The address write-back mode can be selected by use of the <.aa> syntax.

- When using the scaled source addressing mode (.AS), the scale factor is set to 4 for 32-bit Word accesses. The status flags are not updated with this instruction.

Table 7-10 on page 310 lists the address write-back modes.

All prefetch-type instructions are encoded as load instructions with a null destination register (r62), in either of the two 32-bit encodings for loads.

The semantics of each load instruction is governed by four orthogonal bit fields within the instruction format:

- `<zz>` specifies data size: word (00), byte (01), half-word (10) and double-word (11)
- `<aa>` specifies address update/scaling mode: none (00), pre-incr (01), post-incr (10), scaled (11)
- `<d>` specifies cache bypass mode: no-bypass (0), bypass (1)
- `<x>` specifies sign-extension: no sign-extension (0), sign-extension (1)

The `<zz>` and `<aa>` fields are orthogonally available also to prefetch-type instructions, allowing data size and addressing mode to be specified.

The `<x>` and `<d>` fields specify which particular prefetch-type instruction is encoded, when the destination register is r62 (null destination).

Some combinations of values for `<zz>`, `<aa>`, `<d>`, and `<x>` are illegal, although this may also depend on whether the instruction is a load or a prefetch-type.

The PREFETCH, PREFETCHW and PREALLOC instructions are encoded using `<d>` and `<x>` bits, when the destination register is null (r62), as follows:

Table 10-4 PREFETCH Instruction Encodings

Class	<code><d></code>	<code><x></code>	Instruction
PREFETCH	0	0	PREFETCH
PREFETCH	0	1	PREALLOC
PREFETCH	1	0	PREFETCHW
PREFETCH	1	1	Reserved (currently deemed illegal)
Load	-	-	LD, LDB, LDH or LDD, depending on <code><zz></code>

The following combinations of load or prefetch-type modifier fields are illegal, and raise an Illegal Instruction exception.

- The `<x>` and `<d>` bits together select the prefetch-type opcode.
- The '-' character indicates a don't care case.
- The `<aa>` and `<zz>` fields operate identically for both prefetch and load instruction classes. Thus, prefetch-type instructions may have both data-size and address pre/post-increment or scaling, just as load instructions.

All illegal combinations of <d>, <x>, <aa>, and <zz> are shown in [Table 10-5](#). These combinations cover all load and prefetch-type instructions in 32-bit encodings.

Table 10-5 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions

Class	<d>	<x>	<aa>	<zz>	Instruction
PREFETCH	1	1	-	-	Reserved combination of PREFETCH opcode
Load	-	1	-	00	Cannot sign-extend from 32 bits
Load	-	1	-	11	Cannot sign-extend from 64bits
-	-	-	11	01	Cannot scale an address for byte data

Pseudo Code

```

if AA==0 then address = src1 + src2    /* PREFETCH */
if AA==1 then address = src1 + src2
if AA==2 then address = src1
if AA==3 then
    address = src1 + (src2 << 2)
if AA==1 or AA==2 then
    src1 = src1 + src2
DEBUG[LD] = 1

if NoFurtherLoadsPending() then      /* On Returning Load */
    DEBUG[LD] = 0

```

Assembly Code Example

```

PREFETCH [r1,4]    ; Prefetch the cache line from memory containing
                    address r1+4

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

PREFETCH<.aa>	[b, c]	00100bbb ^{aa} 110RR00BBBCCCCC111110
PREFETCH<.aa>	[b, s9]	00010bbbssssssssSBBB0aaRR0111110

PUSH

Function

Synonym for [ST ST STH STB](#).

Syntax Example

`PUSH %r0;` this instruction is equivalent to `ST.AW %r0, [%sp, -4];`

PUSH_S

Function

Push onto Stack

Extension Group

BASELINE

Operation

$SP = SP - 4$; $*SP = b$;

Instruction Format

op b

Syntax Example

PUSH_S b

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Decrement 4 bytes from the implicit stack pointer address SP (28) and perform a 32-bit word memory write to that address with the data specified in the source operand (b).

The status flags are not updated with this instruction.

This instruction is equivalent to a store instruction [ST ST STH STB](#) when used with the following syntax:

ST.AW c,[SP,-4]

Pseudo Code

```
SP = SP - 4          /* PUSH_S */
Memory(SP, 4) = src
```

Assembly Code Example

```
PUSH_S r1      ; Subtract 4 from SP and then store the word in r1 to memory  
               ;at address SP
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

PUSH_S	b	11000	bbb	111	00001
PUSH_S	blink	11000	RRR	111	10001

RCMP

Function

Reverse Comparison

Extension Group

BASELINE

Operation

if (cc) $c - b$;

Instruction Format

op b,c

Syntax Example

RCMP<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

A reverse comparison is performed by subtracting source operand 1 (B) from source operand 2 (C) and subsequently updating the flags.

There is no destination register. Therefore, the result of the subtract is discarded.



Note

RCMP always sets the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                                /* RCMP */
  alu = src2 - src1
  Z_flag = if alu==0 then 1 else 0
  N_flag = alu[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
RCMP r1,r2    ; Subtract r1 from r2 and set the flags on the result
```

Syntax and Encoding

RCMP always set the flags even though there is no associated flag setting suffix.

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
RCMP	b, c	00100bbb0000011011BBBCCCCCRRRRRR
RCMP	b, u6	00100bbb0100011011BBBuuuuuuRRRRRR
RCMP	b, s12	00100bbb1000011011BBBssssssSSSSSS
RCMP<.cc>	b, c	00100bbb1100011011BBBCCCCC0QQQQQ
RCMP<.cc>	b, u6	00100bbb1100011011BBBuuuuuu1QQQQQ

REM

Function

2's complement integer Remainder. For the DSP version, see [REM](#).

Extension Group

DIV_REM_OPTION!=none or DSP_DIVSQRT!=none

Operation

if (cc) then $a = b \% c$;

Instruction Format

op a, b, c

Syntax Example

REM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if divisor is zero or if an arithmetic overflow occurs

Description

If the divisor (c) is non-zero, the destination register is assigned the remainder obtained by signed integer division of source operand (b) by source operand (c).

If the divisor (c) is zero, the destination register is never updated.

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

An arithmetic overflow is signaled if the quotient of (b / c) cannot be represented in 32 bits. This overflow only occurs in one specific case, which is the division of the largest representable negative value (0x80000000) by -1. If an arithmetic overflow occurs, the overflow flag (V) is set to 1.

If a division-by-zero is attempted, or if the result cannot be represented in 32 bits, when the EV_DivZero exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, “Implementation Behavior on Division Overflow”](#).

Pseudo Code

```

if (cc == true) {
    if ((src2 != 0) && ((src1 != 0x80000000) || (src2 != 0xffffffff)))
    {
        dest = src1 % src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = dest[31];
            V = 0;
        }
    } else {
        if ((src2 == 0) && (STATUS32.DZ == 1))
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1){
                V = 1;
            }
        }
    }
}
}

```

Assembly Code Example

```
REM r1,r2,r3                ; r1 is assigned r2 % r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

REM<.f>	a,b,c	00101bbb00001000FBBBCCCCC	AAAAAA
REM<.f>	a,b,u6	00101bbb01001000FBBBuuuuuu	AAAAAA
REM<.f>	b,b,s12	00101bbb10001000FBBBssssss	SSSSSS
REM<.cc><.f>	b,b,c	00101bbb11001000FBBBCCCCC	0QQQQQ
REM<.cc><.f>	b,b,u6	00101bbb11001000FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#)

REMU

Function

Unsigned integer Remainder. For the DSP version, see [REMU](#).

Extension Group

DIV_REM_OPTION!=none or DSP_DIVSQRT!=none

Operation

if (cc) then $a = b \% c$;

Instruction Format

op a, b, c

Syntax Example

REMU <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always cleared
C		= Unchanged
V	•	= Set if divisor is zero

Description

If the divisor (c) is non-zero, the destination register is assigned the remainder obtained by unsigned integer division of source operand (b) by source operand (c).

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

If a division-by-zero is attempted, when the EV_DivZero exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, "Implementation Behavior on Division Overflow"](#).

If the flag-update bit is set, an unsigned REMU operation always clears the N-flag.

Pseudo Code

```
if (cc == true) {
    if (src2 != 0) {
        dest = src1 % src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = 0;
            V = 0;
        }
    } else {
        if (STATUS32.DZ == 1)
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1){
                V = 1;
            }
        }
    }
}
```

Assembly Code Example

```
REMU r1,r2,r3                ; r1 is assigned r2 % r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
REMU<.f>	a,b,c	00101bbb00001001FBBBCCCCCAAAAAA
REMU<.f>	a,b,u6	00101bbb01001001FBBBuuuuuuAAAAAA
REMU<.f>	b,b,s12	00101bbb10001001FBBBssssssSSSSSS
REMU<.cc><.f>	b,b,c	00101bbb11001001FBBBCCCCC0QQQQQ
REMU<.cc><.f>	b,b,u6	00101bbb11001001FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

RLC

Function

Rotate Left Through Carry

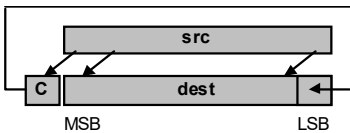
Extension Group

BASELINE

Operation

$$b=c<<1; b=b \mid C; C=c>>31;$$

Note: In this instruction, C represents the carry bit.



Instruction Format

op b,c

Syntax Example

RLC<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Undefined

Description

Rotate the source operand (c) left by one and place the result in the destination register (b).

The carry flag is shifted into the least-significant bit of the result, and the most-significant bit of the source is placed in the carry flag.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

dest = src << 1                                /* RLC */
dest[0] = C_flag
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[31]
  V_flag = UNDEFINED

```

Assembly Code Example

```

RLC r1,r2    ; Rotate left through carry contents of r2 by one bit and
              ;write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

RLC<.f>	b, c	00100bbb00101111FBBBCCCCC001011
RLC<.f>	b, u6	00100bbb01101111FBBBuuuuuu001011

ROL

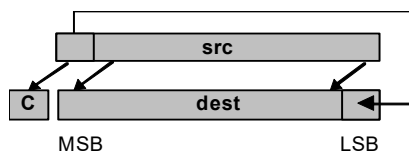
Function

Rotate Left

Extension Group

BASELINE

Operation

$$b = (c \ll 1) \mid (c \gg 31);$$


Instruction Format

op b,c

Syntax Example

ROL<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set to most-significant bit of source operand
V		= Unchanged

Description

Rotate the source operand (c) 1 bit to the left and place the result in the destination register (b).

The most-significant bit of the source operand is copied to the carry flag.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (src << 1) | (src >> 31)          /* ROL */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[31]
```

Assembly Code Example

```
ROL r1,r2    ; Rotate r2 1 place to the left, writing the result to r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

ROL<.f>	b,c	00100	bbb00	101111	FBBB	CCCCC	001101
ROL<.f>	b,u6	00100	bbb01	101111	FBBB	uuuuuu	001101

ROL8

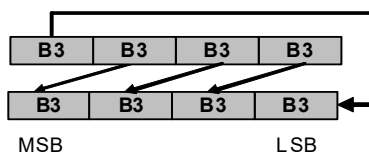
Function

Rotate Left 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$$b = (c \ll 8) \mid (c \gg 24);$$


Instruction Format

op b,c

Syntax Example

ROL8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Rotate the source operand 8 places to the left.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (src << 8) | (src >> 24)          /* ROR8 */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
ROL8 r1,r2    ; Rotate r2 8 places to the left, placing
               ; result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

ROL8<.f>	b,c	00101bbb00101111FBBBCCCCC010000
ROL8<.f>	b,u6	00101bbb01101111FBBBuuuuuu010000

ROR

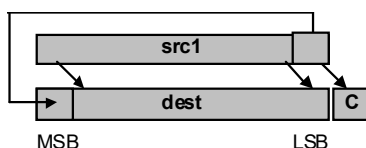
Function

Rotate Right

Extension Group

BASELINE

Operation

$$C = c \ \& \ 1; \ b = c \gg 1 \mid c \ll 31;$$


Instruction Format

op b,c

Syntax Example

ROR<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Rotate the source operand (c) right by one and place the result in the destination register (b).

The least-significant bit of the source operand is copied to the carry flag.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

dest = src >> 1                                /* ROR */
dest[31] = src[0]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[0]

```

Assembly Code Example

```

ROR r1,r2    ; Rotate right contents of r2 by one bit and write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ROR<.f>	b,c	00100bbb00101111FBBBCCCCC000011
ROR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000011

ROR multiple

Function

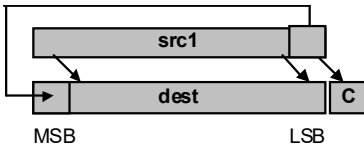
Multiple Rotate Right

Extension Group

SHIFT_OPTION 2 or 3

Operation

if (cc) {C = b >> (c-1) & 1; a = b >> c | b << (31-c);}



Instruction Format

op a, b, c

Syntax Example

ROR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Rotate operand (b) right by (c) places and place the result in the destination register a. Only the bottom 5 bits of (c) are used as the shift value.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc=true then                                     /* ROR */
  dest = src1 >> (src2 & 31)                       /* Multiple */
  dest [31:(31-src2)] = src1 [(src2-1):0]
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = if src2==0 then 0 else src1[src2-1]

```

Assembly Code Example

```
ROR r1,r2,r3 ;Rotate right contents of r2 by r3 bits and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ROR<.f>	a,b,c	00101bbb00000011FBBBCCCCC	AAAAAA
ROR<.f>	a,b,u6	00101bbb01000011FBBBuuuuuu	AAAAAA
ROR<.f>	b,b,s12	00101bbb10000011FBBBssssss	SSSSSS
ROR<.cc><.f>	b,b,c	00101bbb11000011FBBBCCCCC	0QQQQQ
ROR<.cc><.f>	b,b,u6	00101bbb11000011FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ROR8

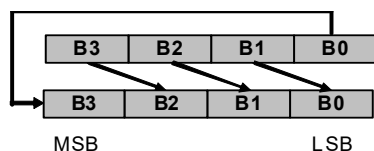
Function

Rotate right 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$$b = (c \gg 8) \mid (c \ll 24);$$


Instruction Format

op b,c

Syntax Example

ROR8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Rotate the source operand 8 places to the right.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (src >> 8) | (src << 24)          /* ROR8 */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
ROR8 r1,r2    ; Rotate r2 8 places to the right, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

```
ROR8<.f>    b,c      00101bbb00101111FBBBCCCCC010001
```

```
ROR8<.f>    b,u6     00101bbb01101111FBBBuuuuuu010001
```

RRC

Function

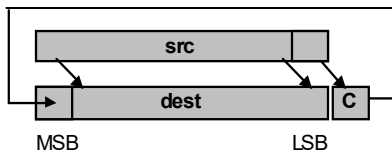
Rotate Right through Carry

Extension Group

BASELINE

Operation

$b = c \gg 1 \mid C \ll 31; C = c \& 1;$



Instruction Format

op b,c

Syntax Example

RRC<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Rotate the source operand (c) right by one and place the result in the destination register (a).

The carry flag is shifted into the most-significant bit of the result, and the least-significant bit of the source is placed in the carry flag.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 1                                /* RRC */
dest[31] = C_flag
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[0]
```

Assembly Code Example

```
RRC r1,r2    ; Rotate right through carry contents of r2 by one bit and
              ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

RRC<.f>	b, c	00100bbb00101111FBBBCCCCC000100
RRC<.f>	b, u6	00100bbb01101111FBBBuuuuuu000100

RSUB

Function

Reverse Subtract

Extension Group

BASELINE

Operation

if (cc) $a = c - b$;

Instruction Format

op a, b, c

Syntax Example

RSUB<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Subtract source operand 1 (b) from source operand 2 (c) and place the result in the destination register (a).

If the carry flag is set upon performing the subtract, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* RSUB */
    dest = src2 - src1
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
RSUB r1,r2,r3    ; Subtract contents of r2 from r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
RSUB<.f>	a,b,c	00100bbb00001110FBBBCCCCCAAAAA
RSUB<.f>	a,b,u6	00100bbb01001110FBBBuuuuuuAAAAA
RSUB<.f>	b,b,s12	00100bbb10001110FBBBssssssSSSSSS
RSUB<.cc><.f>	b,b,c	00100bbb11001110FBBBCCCCC0QQQQQ
RSUB<.cc><.f>	b,b,u6	00100bbb11001110FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

RTIE

Function

Return from Interrupt/Exception

Extension Group

BASELINE

Operation

```
if ((HAS_INTERRUPTS == 0) || (STATUS32.AE) || (AUX_IRQ_ACT[15:0] == 0))
    ReturnFromException()
else
    if ((AUX_IRQ_ACT[0]) && (FIQ_OPTION))
        if (IRQ_PRIORITY_PENDING[0])
            ServiceAnotherFastInterrupt()
        else
            ReturnFromFastInterrupt()
    else
        if (PendingNewIntrPreemptsReturnLevel)
            ServiceAnotherInterrupt()
        else
            ReturnFromInterrupt()
```



Note

For information about the conditions and function used in this section, see the sections: [Fast Interrupt Exit](#), [Returning from Regular Interrupts](#), [Exception Exit](#), and [Exceptions and Delay Slots](#).

Instruction Format

Zero operands

Syntax Example

RTIE

STATUS32 Flags Affected

IE		• = Set according to status register update
AD		• = Set according to status register update
RB[2:0]		• = Set according to status register update
ES		• = Set according to status register update
SC		• = Set according to status register update
DZ		• = Set according to status register update
L		• = Set according to status register update

Z	•	= Set according to status register update
N	•	= Set according to status register update
C	•	= Set according to status register update
V	•	= Set according to status register update
U	•	= Set according to status register update
DE	•	= Set according to status register update
AE	•	= Set according to status register update
E[3:0]	•	= Set according to status register update
H	•	= 0

Description

The return from interrupt or exception instruction, RTIE, allows exit from interrupt and exception handlers, and also allows the processor to switch from kernel mode to user mode.

The RTIE instruction is available only in kernel mode. Using this instruction in the user mode raises a [Privilege Violation, Kernel/User Only Access](#) exception.

The interrupt and exception handlers use the RTIE to exit an interrupt or exception. The RTIE instruction restores previous context including the program counter, status register and, optionally, selected core registers depending on whether the return is from an exception, a fast or regular interrupt, and to what machine operating level the processor is returning. If an RTIE is executed on a processor which has no interrupts configured, the RTIE performs a return from exception as illustrated in the pseudo-code function, `ReturnFromException()`.

Bits in the STATUS32, AUX_IRQ_ACT and IRQ_PRIORITY_PENDING registers are provided to allow the RTIE instruction to determine what pre-interrupt or exception machine state to restore and from where to reload the state. Machine state is restored from ERET, ERSTATUS, and ERBTA registers when returning from an exception, from ILINK, STATUS32_P0 registers when returning from a fast interrupt and, from a user stack or kernel stack when returning from a regular interrupt. When returning from a regular interrupt, selected core and auxiliary registers may also be restored from the stack depending on bits in the AUX_IRQ_CTRL register.

When the core is executing a microcoded epilog sequence of an RTIE, accesses the stack in memory, it is possible that an exception such as a memory protection violation or stack checking violation may occur. So, the epilog sequence is designed to be re-startable such that the stack pointer is not modified until an epilog sequence is successfully completed.

On return from a fast interrupt the processor restores the STATUS32 register including the RB field. When more than one bank of core registers are configured, the processor implicitly switches to the register bank defined by the RB field in the restored STATUS32 register. A detailed discussion of the actions taken on interrupt and exception entry and exit is available in sections:

- [Fast Interrupt Entry](#)
- [Fast Interrupt Exit](#)

- ❑ [Regular Interrupt Entry](#)
- ❑ [Returning from Regular Interrupts](#)
- ❑ [Exception Entry](#)
- ❑ [Exception Exit](#)
- ❑ [Exceptions and Delay Slots](#)

As exceptions are permitted between a branch/jump and an executed delay slot instruction, special branch target address registers are used for exception handler returns.

If the STATUS32[DE] bit is set as a result of the RTIE instruction, the processor is put back into a state where a branch with a delay slot is pending. The target of the branch is contained in the BTA register which is restored from the appropriate Exception Return BTA register (ERBTA).

Pseudo Code Example

To understand the pseudo code, refer to the following notation:

- ❑ P is the processor's current operating priority defined by the index of least-significant bit set (with value of 1) in AUX_IRQ_ACT.ACTIVE. P is 16 if no interrupts are active.
- ❑ Q is the index of the least-significant bit set (with a value of 1) in AUX_IRQ_ACT.ACTIVE when bit P is cleared. Q is 16 if P is the only bit set to 1 in AUX_IRQ_ACT.ACTIVE.
- ❑ PI is the highest priority pending and enabled interrupt, and its priority is W.
- ❑ If the AE bit is set in STATUS32, or if there is no active interrupt handler, an RTIE instruction assumes a return from exception.
- ❑ If the AE bit is not set, and if there is an active interrupt handler, an RTIE instruction assumes a return from interrupt. The behavior of a return from interrupt depends on PI and W, that is, the next pending interrupt and its priority.

```
ReturnFromInterruptOrException()
if ((HAS_INTERRUPTS == 0) || (STATUS32.AE == 1) || (P == 16))
    ReturnFromException()
else
    if ((P == 0) && (FIRQ_OPTION == 1))
        ReturnFromFastInterrupt (PI, W)
    else
        ReturnFromInterrupt (PI, W)

ReturnFromException()
Next_PC = ERET
STATUS32 = ERSTATUS
BTA = ERBTA
if (ERSTATUS.U)                                //check for User mode
    AEX R28, [AUX_USER_SP]                      //restore User SP
Jump (Next_PC)
```

Assembly Code Example

```
RTIE                                ; Return from interrupt/exception
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
RTIE    00100100011011110000RRRRRR111111
```

SBC

Function

Subtract with Carry

Extension Group

BASELINE

Operation

if (cc) $a = (b - c) - C$;

Instruction Format

op a, b, c

Syntax Example

SBC<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Subtract source operand 2 (c) from source operand 1 (b) and also subtract the state of the carry flag C. If C is set, subtract 1; otherwise, subtract 0. Place the result in the destination register a.

If the carry flag is set upon performing the subtract, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                /* SBC */
  dest = (src1 - src2) - C_flag
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = Overflow()
```


Assembly Code Example

```
SBC r1,r2,r3    ; Subtract with carry contents of r3 from r2 and write
                ;result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SBC<.f>	a,b,c	00100bbb00000011FBBBCCCCCAAAAAA
SBC<.f>	a,b,u6	00100bbb01000011FBBBuuuuuuAAAAAA
SBC<.f>	b,b,s12	00100bbb10000011FBBBssssssSSSSSS
SBC<.cc><.f>	b,b,c	00100bbb11000011FBBBCCCCC0QQQQQ
SBC<.cc><.f>	b,b,u6	00100bbb11000011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SETcc

Function

Compute Boolean relation between operands

Extension Group

CODE_DENSITY

Operation

$\text{dest} \leftarrow \text{Relation}(\text{src1}, \text{src2})$

Instruction Format

op a, b, c

Syntax Example

SETcc<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if both source operands are equal
N	•	= Set if (src1 – src2) is negative
C	•	= Set if carry is generated by (src1 – src2)
V	•	= Set if an overflow is generated by (src1 – src2)

Description

Compare the two signed source operands (b) and (c) using the selected relational operator, and place the Boolean result (0 or 1) in the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* SETcc */
dest = (src1 Relop src2)
  if F==1 then
    Z_flag = if (src1 == src2) then 1 else 0
    N_flag = if (src1 < src2) then 1 else 0
    V_flag = Overflow (src1-src2)
    C_flag = CarryOut (src1-src2)

```

Assembly Code Example

```
SETLT r1,r2,r3    ; Set r1 to 1 if r2 < r3 otherwise set r1 to 0
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SETcc<.f>	a,b,c	00100bbb00iiiiiiiFBBBCCCCCAAAAAA
SETcc<.f>	a,b,u6	00100bbb01iiiiiiiFBBBuuuuuuAAAAAA
SETcc<.f>	b,b,s12	00100bbb10iiiiiiiFBBBssssssSSSSSS
SETcc<.cc><.f>	b,b,c	00100bbb11iiiiiiiFBBBCCCCC0QQQQQ
SETcc<.cc><.f>	b,b,u6	00100bbb11iiiiiiiFBBBuuuuuu1QQQQQ

SETcc encodings

Sub-opcode	iiii	Instruction mnemonic	Description	Operand type
0x38	111000	SETEQ	set if equal	Signed or unsigned
0x39	111001	SETNE	set if not equal	Signed or unsigned
0x3a	111010	SETLT	set if less than	Signed
0x3b	111011	SETGE	set if greater or equal	Signed
0x3c	111100	SETLO	set if lower than	Unsigned
0x3d	111101	SETHS	set if higher or same	Unsigned
0x3e	111110	SETLE	set if less than or equal	Signed
0x3f	111111	SETGT	set if greater than	Signed

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SETI

Function

SETI enables interrupts, and if a source operand is mentioned, sets the interrupt threshold in the STATUS32 register.

Extension Group

HAS_INTERRUPTS == 1

Operation

```
{
  if (c[5] == 1)
    STATUS32.E ← [3:0]
    STATUS32.IE ← c[4]
  else
    STATUS32.IE ← 1
    if (c[4] == 1)
      STATUS32.E ← c[3:0]
}
```

Instruction Format

op c

Syntax Example

SETI c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

SETI takes a single u6 or C source register operand that indicates whether the interrupt priority level is set when enabling interrupts, and if so to what level.

When an ARCV2-based processor is configured without interrupts (HAS_INTERRUPTS = 0), executing a SETI instruction raises an [Illegal Instruction](#) exception.

The SETI instruction is available only in kernel mode. Using this instruction in the user mode raises a [Privilege Violation, Kernel/User Only Access](#) exception.

If your processor includes the SecureShield component,

- the SETI instruction is available only in the secure kernel mode if SEC_STAT.NIC==0. Usage of this instruction in any other mode raises a Privilege Violation exception (0x071000).
- the SETI instruction is available in the secure kernel and normal kernel modes if SEC_STAT.NIC==1.
- Using the SETI instruction in the secure or normal user mode raises a Privilege Violation exception (0x071020).

Pseudo Code

```

if (SEC_MODES_OPTION == 1)/*SecureShield is present*/           /*SETI*/
{
    if ((STATUS32.U == 0) && ((STATUS32.S == 1) || (SEC_STAT.NIC ==
1)))
    {
        if (c[5] == 1)
            STATUS32.E = c[3:0]
            STATUS32.IE = c[4]
        else
            STATUS32.IE = 1
        if (c[4] == 1)
            STATUS32.E = c[3:0]
    }
}
else /* SecureShield is not present */
{
    if (c[5] == 1)
        STATUS32.E = c[3:0]
        STATUS32.IE = c[4]
    else
        STATUS32.IE = 1
    if (c[4] == 1)
        STATUS32.E = c[3:0]
}

```

Assembly Code Example

```

SETI R0    ;enable interrupts and restore the preempting interrupt
           ;priority value.

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```

SETI          c          00100110001011110000CCCCC111111

```

SETI u6 00100110011011110000uuuuuu111111

SETI takes a 6-bit operand. Bit 5 indicates whether interrupt enables are being restored (1) or forced to specific values (0).

When forcing the interrupt enable to 1, the STATUS32.E field can be optionally set according to bits [3:0] of the SETI operand.

Bit5	Bit4	Purpose
0	0	Enable interrupts without changing the enabled level
0	1	Enable interrupts and also set the enabled level
1	x	Restore STATUS32.IE and STATUS32.E using bits 4 and [3:0] respectively

The assembler must encode a SETI without operands as SETI u6, with u6 == 0 to provide a concise way of specifying “enable interrupts”.

SETI is a serializing instruction similar to the KFLAG instruction. This behavior means after this instruction commits, the pipeline is flushed.

SEXB

Function

Sign Extend Byte

Extension Group

BASELINE

Operation

$$b = (c \ll 24) \gg 24;$$

Instruction Format

op b,c

Syntax Example

SEXB<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Sign extend the byte contained in the least-significant 8 bits of source operand (c) to a full 32-bit word value and place the result into the destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest[7:0] = src[7:0]           /* SEXB */
dest[31:8] = src[7]
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```


Assembly Code Example

```
SEXB r1,r2    ; Sign extend the bottom 8 bits of r2 and write result to r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SEXB<.f>	b,c	00100bbb00101111FBBBCCCCC000101
SEXB<.f>	b,u6	00100bbb01101111FBBBuuuuuu000101
SEXB_S	b,c	01111bbbccc01101

SEXH ~~SEXW~~

Function

Sign Extend Half-word 16-bits to word



Note

The SEXW mnemonic is deprecated.

Extension Group

BASELINE

Operation

$$b = (c \ll 16) \gg 16;$$

Instruction Format

op b,c

Syntax Example

SEXH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Sign extend the 16-bit half-word contained in the least-significant 16 bits of source operand (c) to a full 32-bit word value and place the result into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest[15:0] = src[15:0]           /* SEXH */
dest[31:16] = src[15]
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
SEXH r1,r2    ; Sign extend the bottom 16 bits of r2 and write result to r1
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code

SEXH<.f>	b,c	00100bbb00101111FBBBCCCCC000110
SEXH<.f>	b,u6	00100bbb01101111FBBBuuuuuu000110
SEXH_S	b,c	01111bbbccc01110

SFLAG

Function

Set secure status flags

Extension Group

SecureShield

Operation

Assign bits from the operand to the SEC_STAT register.

Instruction Format

op c

You cannot use the XY operands as a source or destination with this instruction.

Syntax Example

SFLAG c

STATUS32 Flags Affected

Flag	Mode	Source of operand
NIC	• Secure kernel	Bit 5 of Source Operand
SUE		Unchanged
IRM	• Secure kernel	Bit 3 of Source Operand
NSRU	• Secure user or kernel	Bit 2 of Source Operand
NSRT	• Secure user or kernel	Bit 1 of Source Operand
SSC	• Secure kernel	Bit 0 of Source Operand

Description

The contents of the source operand (c) are used to set the flags held in the processor secure status register, SEC_STAT.

The SFLAG instruction is serializing – ensuring that no further instructions can be completed before all flag updates take effect.

This instruction can be executed in either secure modes (kernel or user), execution of this instruction in normal user or kernel mode results in an EV_PrivilegeV (ECR = 0x071020) exception. This

instruction is only available `-sec_modes_option == 1`. If `-sec_modes_option == 0`, and you use this instruction, an illegal instruction exception (0x020000) is raised.

Pseudo Code

```
if (STATUS32.U == 0) /*in kernel mode*/           /* SFLAG */
    if (STATUS32.S == 1)/*in secure mode*/
        SEC_STAT.SSC = src[0]/* SSC */
SEC_STAT.IRM = src[3]/* IRM */
        SEC_STAT.NIC = src[5]/* NIC */
If (STATUS32.S == 1)/*in secure mode*/
SEC_STAT.NSRU = src[2]/* NSRU */
SEC_STAT.NSRT = src[1]/* NSRT */
```

Assembly Code Example

```
SFLAG 1                                ; Enable secure stack checking
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SFLAG	c	00110000001011110000CCCCC111111
SFLAG	u6	00110000011011110000uuuuuu111111

SLEEP

Function

Enter Sleep State

Extension Group

BASELINE

Operation

DEBUG[ZZ] = 1;

Instruction Format

op c

Syntax Example

SLEEP

STATUS32 Flags Affected

IE	Updated depending on SLEEP operand as explained in Table 10-6
E[3:0]	Updated depending on SLEEP operand as explained in Table 10-6
Z	= Unchanged
N	= Unchanged
C	= Unchanged
V	= Unchanged
ZZ	= 1
SM	Updated depending on SLEEP operand as explained in Table 10-6

Description

The ARCV2-based processor enters the sleep state when the processor encounters the SLEEP instruction while operating in a kernel mode. The processor stays in the sleep state until an interrupt or restart occurs. Power consumption may be reduced during sleep state because the processor is inactive, and the RAMs may be disabled. The SLEEP instruction is available only in kernel mode. Using this instruction in the user mode raises a [Privilege Violation, Kernel/User Only Access](#) exception.

If SEC_STAT.NIC==0, the SLEEP instruction is available in the normal and secure kernel modes. However, in the normal kernel mode, you can only set the STATUS32.IE bit only; Writes to the STATUS32.E are ignored.

If SEC_STAT.NIC==1, the SLEEP instruction is available in the normal kernel and the secure kernel mode. You can set the STATUS32.IE and STATUS32.E fields.

SLEEP instruction operands

The SLEEP instruction is a single operand instruction without flags. A SLEEP instruction without a source operand is encoded as SLEEP 0.

The SLEEP instruction is serializing, which means the SLEEP instruction completes and then flushes the pipeline.

In sleep state, the sleep state flag (ZZ) is set, and the host interface operates in the normal way, allowing access to the DEBUG and the STATUS registers. The host interface also has the ability to halt the processor. The host cannot clear the sleep state flag, but the host can wake the processor by halting the processor, and then restarting it. The program counter (PC) points to the next instruction in sequence after the sleep instruction.

The processor wakes up from sleep state when an interrupt occurs, a non-maskable imprecise exception is triggered, or when the processor is restarted. If an interrupt wakes the processor, the ZZ flag is cleared, the interrupt routine is serviced, and execution resumes at the instruction in sequence after the SLEEP instruction. When a processor is started after being halted, the ZZ flag is cleared.

The SLEEP instruction is a NOP during the single-step mode. Consequently, the behavior is of NOP instruction and the IE, E[3:0], ZZ & SM bits are unchanged.

Every single-step operation is a restart and the ARCV2-based processor wakes up at the next single-step.

Three operand bits ([7:5]) of the SLEEP instruction indicate to the system the specific sleep mode through the sys_sleep_mode_r[2:0] output vector.

Figure 10-2 SLEEP Operand

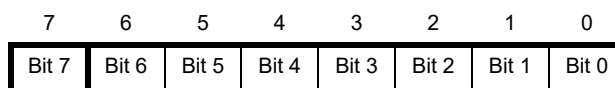


Table 10-6 SLEEP Operand

Bits	Description
Bit[3:0]	Processor interrupt threshold value. When SLEEP.[4] bit is set to 1, the instruction updates STATUS32.E[3:0] bits with these bits. Any interrupt with a higher priority than this interrupt threshold awakens the processor.
Bit 4	Enable interrupts. When this bit is set to 1, interrupts are enabled and the processor updates the STATUS32.IE bit to 1.
Bit [7:5]	User-defined sleep mode status.

The bottom 5 bits of the source field, u6 or c, are used as the enable flags value, and the remaining 1 (u6 operand) or 3 bits (c operand) are used only for setting the SYS_SLEEP_MODE outputs.

Pseudo Code

```
(if c register operand...)                                /* SLEEP */

FlushPipe()
DEBUG[ZZ] = 1
DEBUG[SM] = c[7:5]
SYS_SLEEP_MODE[2:0] = c[7:5]
if (c[4] == 1)
{
    STATUS32.E = c[3:0]
    STATUS32.IE = 1
}
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()

(if u6 immed operand...)

FlushPipe()
DEBUG[ZZ] = 1
DEBUG [SM] = {2'b00,u[5]}
SYS_SLEEP_MODE[2:0] = {2'b00,u[5]}
if (u[4] == 1)
{
    STATUS32.E = u[3:0]
    STATUS32.IE = 1
}
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()
```



```

if (SEC_MODES_OPTION == 1)/*SecureShield is present*/                                /*SLEEP*/
if ((STATUS32.U == 0) && ((STATUS32.S == 1) || (SEC_STAT.NIC == 1)))
/*in secure kernel mode or normal kernel mode with NIC == 1*/
(if c register operand...)
FlushPipe()
DEBUG[ZZ] = 1
DEBUG[SM] = c[7:5]
SYS_SLEEP_MODE[2:0] = c[7:5]
if (c[4] == 1)
{
STATUS32.E = c[3:0]
STATUS32.IE = 1
}
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()
(if u6 immed operand...)
FlushPipe()
DEBUG[ZZ] = 1
DEBUG [SM] = {2'b00,u[5]}
SYS_SLEEP_MODE[2:0] = {2'b00,u[5]}
if (u[4] == 1)
{
STATUS32.E = u[3:0]
STATUS32.IE = 1
}
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()

else if ((STATUS32.S == 0) && (STATUS32.U == 0) && (SEC_STAT.NIC == 0))
/*in normal kernel mode with NIC == 0*/
(if c register operand...)
FlushPipe()
DEBUG[ZZ] = 1
DEBUG[SM] = c[7:5]
SYS_SLEEP_MODE[2:0] = c[7:5]
if (c[4] == 1)
{
STATUS32.IE = 1
}
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()
(if u6 immed operand...)
FlushPipe()
DEBUG[ZZ] = 1
DEBUG [SM] = {2'b00,u[5]}
SYS_SLEEP_MODE[2:0] = {2'b00,u[5]}
if (u[4] == 1)
{
STATUS32.IE = 1
}
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()

```

```

else  /* SecureShield is not present */
(if c register operand...)
FlushPipe()
DEBUG[ZZ] = 1
DEBUG[SM] = c[7:5]
SYS_SLEEP_MODE[2:0] = c[7:5]
if (c[4] == 1)
{
STATUS32.E = c[3:0]
STATUS32.IE = 1
}
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()
(if u6 immed operand...)
FlushPipe()
DEBUG[ZZ] = 1
DEBUG [SM] = {2'b00,u[5]}
SYS_SLEEP_MODE[2:0] = {2'b00,u[5]}
if (u[4] == 1)
{
STATUS32.E = u[3:0]
STATUS32.IE = 1
}
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()

```

Assembly Code Example

The SLEEP instruction can be placed anywhere in the instruction sequence.

Example 10-5 Sleep placement in code

```

SUB r2, r2, 0x1
ADD r1, r1, 0x2
SLEEP
...

```

[Example 10-6](#) illustrates the use of the SLEEP instruction with each combination of interrupts levels enabled or disabled during sleep.

Example 10-6 Enable Interrupts and Sleep

SLEEP	;enter sleep state without changing interrupt enable or ;interrupt priority level
SLEEP 0x04	;enter sleep state without changing interrupt enable or interrupt priority ;level
SLEEP 0x014	;enter sleep and enable interrupts and sets interrupt ;priority level to 4. Any interrupt with a priority ;equal to or higher than 4 awakens the ;processor.

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SLEEP	c	00100	001	00	101111	0	000	CCCCC	111111
SLEEP	<u6>	00100	001	01	101111	0	000	uuuuuu	111111

SJLI

Function

Secure Jump and Link Indirect

Extension Group

`-sec_modes_option==true`

Operation

`BLINK = next_PC; PC = NSC_TABLE_BASE + (u12);`

Instruction Format

`op u12`

Syntax Example

`SJLI u12`

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Secure mode functions can be registered as normal callable at compile time. This registration causes an entry to be added to a special jump table (defined by `NSC_TABLE_BASE` and `NSC_TABLE_TOP` registers) in the secure memory space containing the start address of that function. The call is then implemented in normal code using the SJLI instruction that references the relevant offset in that table. The return address in the normal code is stored in the BLINK register as normal.

The SJLI index bounds are checked and when an out of range occurs, a normal callable function table range violation exception is raised on the last jump micro-operation of the sequence, and the exception PC is the NSC table entry.

If normal code branches and jumps are linearly executed into the secure address space without using the new call instruction, an Instruction Error exception is triggered.

If either the jump table or the target of the location in the jump table are not located in an address range marked as secure in the MPU, an `EV_MachineCheck` exception is triggered when the call is issued.

Pseudo Code

```
BLINK = PC + 2                                /* SJLI */
PC = SJLI + (u12)
```

Assembly Code Example

```
SJLI index      ; store the return address in BLINK, and
                 ; then jump to NSC_TABLE_BASE[index]
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
SJLI  u12  00101RRR101000001RRRuuuuuuUUUUUU
```

SR

Function

Store to Auxiliary Register

Extension Group

BASELINE

Operation

`_sr(b,c);`

Instruction Format

`op b,c`

Syntax Example

`SR b,[c]`

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Store the 32-bit Word held in source operand 1 (b) into the auxiliary register whose address is obtained from the source operand 2 (c).

Pseudo Code

```
Aux_reg(src2) = src1    /* SR */
```

Assembly Code Example

```
SR r1,[r2]    ; Store contents of r1 into auxiliary register pointed by r2
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SR	b, [c]	00100bbb00101011RBBBCCCCCRRRRRR
SR	b, [u6]	00100bbb01101011RBBBuuuuuuRRRRRR
SR	b, [s12]	00100bbb10101011RBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ST ST STH STB

Function

Store to Memory

Extension Group

ST, STH, STB: BASELINE

TY

Operation

addr=b+s9; *addr=c;

Instruction Format

op c,b,s9

Syntax Example

ST<zz><.aa><.di> c,[b,s9]

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Store the data held in source operand 1 (c) in the memory at an address that is calculated by adding source operand 2 (b) and an offset specified by source operand 3 (s9). The status flags are not updated with this instruction.

The size of the data written is specified by the data size field <zz> (32-bit formats).

[Table 7-13](#) on page 312 lists the data size modes.



Note

For more information about the rules governing non-aligned memory references, see [Data Layout in Memory](#).

If the processor contains a data cache, store requests can bypass the cache by using the <.di> syntax.

[Table 7-12](#) on page 311 lists the direct to memory bypass modes.

**Note**

For the 16-bit encoded instructions, the u offset is aligned accordingly. For example ST_S c, [b. u7] only needs to encode the top 5 bits because the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

The address write-back mode can be selected by use of the <.aa> syntax.

Table 7-10 on page 310 lists the address write-back modes.

**Note**

When using the scaled source addressing mode (.AS), the scale factor is dependent upon the size of the requested data word (zz).

Pseudo Code

```
if AA==0 then address = src2 + src3    /* ST */
if AA==1 then address = src2 + src3
if AA==2 then address = src2
if AA==3 and ZZ==0 then
  address = src2 + (src3 << 2)
if AA==3 and ZZ==2 then
  address = src2 + (src3 << 1)
if AA==3 and ZZ==3 then
  address = src2 + (src3 << 2)
Memory(address, size) = src1
if AA==1 or AA==2 then
  src2 = src2 + src3
```

Assembly Code Examples

```
ST r0, [r1,4]    ; Store word value of r0 to memory address r1+4
ST 42, [r1,4]    ; Store immediate 32-bit word 42 to the memory address
                  ;given by r1+4
```

Syntax and Encoding

For more information about the encodings, see “Key for 32-bit Addressing Modes and Encoding Conventions” and “Key for 16-bit Addressing Modes and Encoding Conventions” .

Instruction Code

ST<zz><.aa><.di>	c, [b]	00011bbb000000000BBBCCCCCDDaaZZ0
ST<zz><.aa><.di>	c, [b, s9]	00011bbbssssssssSBBBCCCCCDDaaZZ0
ST<zz><.aa><.di>	w6, [b]	00011bbb000000000BBBwwwwwDaaZZ1
ST<zz><.aa><.di>	w6, [b, s9]	00011bbbssssssssSBBBwwwwwDaaZZ1

Instruction Code

ST_S	b, [SP, u7]	11000bbb010uuuuu
ST_S	c, [b, u7]	10100bbbcccuuuuu
STB_S	b, [SP, u7]	11000bbb011uuuuu

**Note**

The encoding of STB_S b, [SP, u7] includes a 5-bit constant, but this constant gets shifted left by two bit positions, as all stack addresses are expected to be four-byte-aligned. This alignment makes the constant a u7 when considering the range of offsets, although the lower two bits are always 0.

Encodings supported when any CODE DENSITY option is enabled

ST_S	R0, [GP, s11]	01010SSSSSS10sss

Encodings supported when CODE DENSITY option is 2 or 3

STB_S	c, [b, u5]	10101bbbcccuuuuu
STH_S	c, [b, u6]	10110bbbcccuuuuu

Encodings supported by LL64_OPTION

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SUB

Function

Subtract

Extension Group

BASELINE

Operation

if (cc) $a = b - c$

Instruction Format

op a, b, c

Syntax Example

SUB<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Subtract source operand 2 (c) from source operand 1 (b) and place the result in the destination register a.

SUB_S.NE is executed when the Z flag is equal to zero.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.



Note

For the 16-bit encoded instructions that work on the stack pointer (SP), the offset is aligned to 32-bit. For example, SUB_S sp, sp. u7 only needs to encode the top 5 bits because the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

Pseudo Code

```

if cc==true then                                /* SUB */
    dest = src1 - src2
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```

SUB r1,r2,r3      ; Subtract contents of r3 from r2 and write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SUB<.f>	a,b,c	00100bbb0000010FBBBCCCCCAAAAAA
SUB<.f>	a,b,u6	00100bbb01000010FBBBuuuuuuAAAAAA
SUB<.f>	b,b,s12	00100bbb1000010FBBBssssssSSSSSS
SUB<.cc><.f>	b,b,c	00100bbb11000010FBBBCCCCC0QQQQQ
SUB<.cc><.f>	b,b,u6	00100bbb11000010FBBBuuuuuu1QQQQQ
SUB_S	c,b,u3	01101bbbccc01uuu
SUB_S.NE	b,b,b	01111bbb1100000
SUB_S	b,b,c	01111bbbccc00010
SUB_S	b,b,u5	10111bbb011uuuuu
SUB_S	SP,SP,u7	11000001101uuuuu

Encodings supported by the CODE DENSITY options

SUB_S	a,b,c	01001bbbccc10aaa

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SUB1

Function

Subtract with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b - (c \ll 1)$

Instruction Format

op a, b, c

Syntax Example

SUB1<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left shifted by 1) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* SUB1 */
    dest = src1 - (src2 << 1)
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
SUB1 r1,r2,r3    ; Subtract contents of r3 left-shifted one bit from r2
                  ;and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SUB1<.f>	a,b,c	00100bbb00010111FBBBCCCCCAAAAAA
SUB1<.f>	a,b,u6	00100bbb01010111FBBBuuuuuuAAAAAA
SUB1<.f>	b,b,s12	00100bbb10010111FBBBssssssSSSSSS
SUB1<.cc><.f>	b,b,c	00100bbb11010111FBBBCCCCC0QQQQQ
SUB1<.cc><.f>	b,b,u6	00100bbb11010111FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SUB2

Function

Subtract with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b - (c \ll 2)$

Instruction Format

op a, b, c

Syntax Example

SUB2<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left-shifted by 2) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* SUB2 */
    dest = src1 - (src2 << 2)
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```


Assembly Code Example

```
SUB2 r1,r2,r3      ; Subtract contents of r3 leftshifted two
                   ; bits from r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SUB2<.f>	a,b,c	00100bbb00011000FBBBCCCCC AAAAAA
SUB2<.f>	a,b,u6	00100bbb01011000FBBBuuuuuu AAAAAA
SUB2<.f>	b,b,s12	00100bbb10011000FBBBssssss SSSSSS
SUB2<.cc><.f>	b,b,c	00100bbb11011000FBBBCCCCC 0QQQQQ
SUB2<.cc><.f>	b,b,u6	00100bbb11011000FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SUB3

Function

Subtract with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b - (c \ll 3)$

Instruction Format

op a, b, c

Syntax Example

SUB3<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left shifted by 3) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* SUB3 */
  dest = src1 - (src2 << 3)
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = Overflow()

```

Assembly Code Example

```
SUB3 r1,r2,r3    ; Subtract contents of r3 left shifted three
                  ; bits from r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SUB3<.f>	a,b,c	00100bbb00011001FBBBCCCCCAAAAAA
SUB3<.f>	a,b,u6	00100bbb01011001FBBBuuuuuuAAAAAA
SUB3<.f>	b,b,s12	00100bbb10011001FBBBssssssSSSSSS
SUB3<.cc><.f>	b,b,c	00100bbb11011001FBBBCCCCC0QQQQQ
SUB3<.cc><.f>	b,b,u6	00100bbb11011001FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SWAP

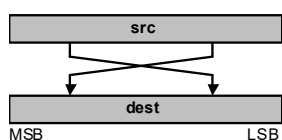
Function

Swap 16-bits half-words

Extension Group

SWAP_OPTION

Operation

$$b = c >> 16 \mid (c \& 0xFFFF) << 16;$$


Instruction Format

op b,c

Syntax Example

SWAP<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Swap the lower 16 bits of the operand with the upper 16 bits of the operand and place the result of that swap in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = SWAP(src)                /* SWAP */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
SWAP r1,r2    ; Swap top and bottom 16 bits of r2 write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

SWAP<.f>	b,c	00101	bbb00	101111	FBBB	CCCCC	000000
SWAP<.f>	b,u6	00101	bbb01	101111	FBBB	uuuuuu	000000

SWAPE

Function

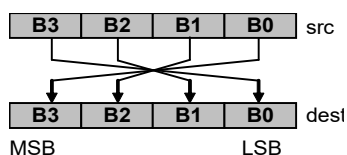
Swap byte ordering

Extension Group

SWAP_OPTION

Operation

b = swap order of bytes from c



Instruction Format

op b,c

Syntax Example

SWAPE<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Copy the source operand to the destination operand, with reversed byte ordering, or Endianness.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (ROL8(src) & 0x00ff00ff)          /* SWAPE */
      | (ROR8(src) & 0xff00ff00)
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
SWAPE r1,r2    ; Take the bytes from r2, in reverse order and copy to r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

SWAPE<.f>	b, c	00101bbb00101111FBBBCCCCC001001
SWAPE<.f>	b, u6	00101bbb01101111FBBBuuuuuu001001

SWI

Function

Software Interrupt or Software Breakpoint

Extension Group

SWI: BASELINE

SWI u6: OS_OPT_OPTION == 1. This option cannot be configured in the ARC EM processor, and therefore this instruction is decoded as an illegal instruction in the ARC EM processor.

Instruction Format

op

Syntax Example

SWI

SWI_S u6

STATUS32 Flags Affected

IE	•	= 0
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
E[3:0]		= Unchanged
U	•	= 0
AE	•	= 1



Note

The SWI_S instruction affects the flags similar to SWI. The above flag status is also valid for SWI_S.

Operation

Software Breakpoint

Description

The SWI instruction provides a means to interrupt software execution at any point in the program. This instruction is typically used by a native debugger to insert soft breakpoints.

The only action performed by a SWI instruction is to raise an EV_SWI exception. The SWI instruction does not advance the program counter before the exception is raised. Therefore, the Exception Return Address register (see [Exception Return Address, ERET](#)) is set to the address of the SWI instruction itself. The Exception Fault Address register (see [Exception Fault Address, EFA](#)) is also set to point to the address of the SWI instruction.

The EV_SWI exception can be raised from user or kernel modes. When inserting a software breakpoint, the instruction at the appropriate address is replaced by a SWI instruction of the same size SWI_S for 16-bit instructions and SWI for 32-bit instructions. Before returning from the EV_SWI exception, the original instruction must replace the SWI or SWI_S instruction. When the exception handler is complete, program execution resumes at the address from which the SWI instruction was executed, and the original instruction executes as normal.

If a source is specified with the SWI_S instruction, the source operand is loaded into the exception cause register (see [Exception Cause Register, ECR](#)) as the cause parameter along with the cause code for a SWI instruction and the SWI vector number, otherwise the parameter field is zero.

The source value can be used to signal a type of command to any operating system that is running on the processor. Source values 0 to 62 can be used to encode distinct operating system calls.

Pseudo Code

```
ERET = currentPC                /* SWI, SWI_S */
ERSTATUS = STATUS32
if STATUS32[DE] == 1 then
    ERBTA = pending PC
ECR = 0x00 : 0x08 : 0x00 : src
EFA = PC
STATUS32[U] = 0
STATUS32[IE] = 0
STATUS32[AE] = 1
PC = INT_VECTOR_BASE + 0x20
```

Assembly Code Example

```
SWI      ; Software interrupt or breakpoint
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SWI	00100	010	01	101111	0	000	RRRRRR	111111
SWI_S	01111	010	111	00000				
SWI_S	n6*	01111	nnnnnn	11111				

*Where n6 != 111111

SYNC

Function

Synchronize

Extension Group

BASELINE

Operation

Wait for all data memory transactions to complete **and previous instructions to retire**

Instruction Format

op

Syntax Example

SYNC

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The synchronize instruction, SYNC, waits until data based memory operations have completed. The SYNC instruction also waits until all previously committed instructions have retired their results.



Note

The SYNC instruction does not wait on memory operations started by other processors.

Use with Interrupts

The SYNC instruction can also be used to ensure that the interrupt request of a memory mapped peripheral has been cleared before an interrupt handler exits.

For example, consider the following scenario:

- ❑ A peripheral generates interrupt to the processor by setting a signal to true.
- ❑ The control registers for the peripheral are memory mapped

- ❑ The processor's interrupt unit is set to 'level sensitive' for this interrupt.
- ❑ The interrupt handler must clear the interrupt request signal before exiting

The SYNC instruction is used to ensure that the store to change the peripheral status happens before the interrupt exit

If the SYNC was not used, the peripheral may still be asserting the interrupt-request signal after the interrupt exit – therefore, an incorrect interrupt is generated.

To provide the instruction synchronize function, the instruction serializes on completion, meaning that the contents of the pipeline are discarded, and fetching restarted from the stored program counter value.

Use for Data Synchronization

For data synchronization, the purpose of the SYNC instruction is to ensure that all the following memory operations started by the processor have finished before any new operations (of any kind) can begin:

- ❑ All outstanding LD, ST and EX instructions
- ❑ All data cache operations, including line fills and flushes

Synchronizing Out-of-order State Updates

Any implementation that permits some instructions to update processor or extension state after the instruction has committed can be forced to wait for all such pending updates by use of the SYNC instruction.

For example, a floating-point extension may provide a multiply operation that commits 2 cycles after the operation is issued, but does not write its result until 3 cycles later. Another example is when a User Extension Instruction is pipelined over a period of 10 cycles, and directly modifies an extension register on completion. On completion of a SYNC instruction, the retirement of any such floating-point results, the update to extension registers, can be guaranteed to have completed for all instructions issued before the SYNC.

Pseudo Code

```
do                                                    /* SYNC */
    null
until not (load_pending or store_pending or dcache_fill or
           dcache_flush_instruction_retirement_pending)
```

Assembly Code Example

```
SYNC                                           ; Synchronize
```

Syntax and Encoding

The status flags are not updated with this instruction. Therefore, the flag setting field, F, is encoded as 0. For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SYNC 00100011011011110000RRRRRR111111

TRAP_S

Function

Trap

Extension Group

BASELINE

Operation

Raise an exception

Instruction Format

op u6

Syntax Example

TRAP_S u6

STATUS32 Flags Affected

IE	•	= 0
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
E[3:0]		= Unchanged
U	•	= 0
AE	•	= 1

Description

The TRAP_S instruction raises an exception and calls any operating system procedures in kernel mode. Traps can be raised from user or kernel modes. The source operand is loaded into the exception cause register (see [Exception Cause Register, ECR](#)) as the cause parameter along with the cause code (0x0) for a trap and the trap vector number.

The source value can be used to signal a type of command to any operating system that is running on the processor. Source values 0 to 63 can be used to encode distinct operating system calls.

If implemented in the processor, the Exception Fault Address register (see [Exception Fault Address, EFA](#)) is set to point to the address of the trap instruction. The Exception Return Address register (see

[Exception Return Address, ERET](#)) is set to the address of the instruction immediately following the trap instruction.

When the exception handler has completed, program execution resumes at the instruction immediately following the trap instruction.

Pseudo Code

```
ERET = NEXTPC                                /* TRAP_S */
ERSTATUS = STATUS32
if STATUS32[DE] == 1 then
    ERBTA = pending PC
ECR = 0x00 : 0x09 : 0x00 : src
EFA = PC
STATUS32[U] = 0
STATUS32[IE] = 0
STATUS32[AE] = 1
PC = INT_VECTOR_BASE + 0x24
```

Assembly Code Example

```
TRAP_S 0                                     ; Trap
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
TRAP_S u6      01111uuuuuu11110
```

TST

Function

Test

Extension Group

BASELINE

Operation

if (cc) b & c;

Instruction Format

op b,c

Syntax Example

TST<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Bitwise AND of source operand 1 (b) with source operand 2 (c) and subsequently update the flags.

There is no destination register. Therefore, the result of the AND is discarded.



Note

TST and TST_S always set the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                                /* TST */
    alu = src1 AND src2
    Z_flag = if alu==0 then 1 else 0
    N_flag = alu[31]
```


Assembly Code Example

```
TST r1,r2    ; Logical AND r2 with r1 and set the flags on the result
```

Syntax and Encoding

The flag setting field, F, is always encoded as 1 for this instruction. For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
TST	b, c	00100bbb000010111BBBCCCCCRRRRRR
TST	b, u6	00100bbb010010111BBBuuuuuuRRRRRR
TST	b, s12	00100bbb100010111BBBssssssSSSSSS
TST<.cc>	b, c	00100bbb110010111BBBCCCCC0QQQQQ
TST<.cc>	b, u6	00100bbb110010111BBBuuuuuu1QQQQQ
TST_S	b, c	01111bbbccc01011

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

UNIMP_S

Function

Unimplemented Instruction

Extension Group

BASELINE

Operation

Raise an [Illegal Instruction](#) exception

Instruction Format

op

Syntax Example

UNIMP_S

STATUS32 Flags Affected

IE	•	= 0
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
E[3:0]		= Unchanged
U	•	= 0
AE	•	= 1

Description

The UNIMP_S instruction always raises an [Illegal Instruction](#) exception. The debugging tools can use this instruction to fill unused memory regions with an instruction that always remains unimplemented, regardless of future additions to the instruction set.

The program counter is not advanced before the exception is raised by this instruction. Therefore, on entry to the exception handler, the ERET register contains the address of the UNIMP_S instruction.

The status flags are not updated by this instruction.

Pseudo Code

```
InstError = 1;    /* UNIMP_S */
```

Assembly Code Example

```
UNIMP_S           ; Unimplemented Instruction
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

UNIMP_S	01111	001	111	00000

WEVT

Function

Enter the sleep state and wait for an event

Extension Group

BASELINE

Operation

DEBUG[ZZ] = 1;

Instruction Format

op c

Syntax Example

WEVT

STATUS32 Flags Affected

IE	•	Updated depending on the SLEEP operand as explained in Table 10-6
E[3:0]	•	Updated depending on the SLEEP operand as explained in Table 10-6
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
ZZ	•	= 1
SM	•	Updated depending on the SLEEP operand as explained in Table 10-6

Description

The ARCV2-based processor enters the sleep state when the processor encounters the WEVT instruction. The processor stays in the sleep state until one of the following events occurs:

- ❑ An external event input is asserted; the external input transitions from 0 to 1.
- ❑ An interrupt is taken.
- ❑ An external halt request is acknowledged by the processor.
- ❑ A debug transaction places the processor in a halt state.

If SEC_STAT.NIC==0, the WEVT instruction is available in the normal and secure kernel modes. However, in the normal kernel mode, you can only set the STATUS32.IE bit only; Writes to the STATUS32.E are ignored.

If SEC_STAT.NIC==1, the WEVT instruction is available in the normal kernel and the secure kernel mode. You can set the STATUS32.IE and STATUS32.E fields.

Using the WEVT instruction in the normal or secure user modes, raises an EV_PrivilegeV exception.

The WEVT instruction is similar to the SLEEP instruction and is available in both kernel and user modes.

- To access the WEVT instruction in user mode, ensure that STATUS32.US == 1.
- If STATUS32.US == 0 and you try to use the WEVT instruction in user mode, the processor raises an EV_PrivilegeV exception.

The WEVT instruction differs from the SLEEP instruction in the following ways:

- The WEVT instruction is available in both kernel and user modes. Whereas, the SLEEP instruction is available only in the kernel mode.
- An external event input when asserted can wake up the processor from an WEVT-induced sleep.

The WEVT instruction has the same operands as the SLEEP instruction. For more information about how the SLEEP instruction operands work, see [“SLEEP instruction operands”](#).

DMA transfers and WEVT instruction

The wait event instruction WEVT can be used to put the processor core into sleep mode when waiting for a DMA transfer to complete. Following the execution of the WEVT instruction, the gated clocks of the core are disabled and the ZZ bit is set in the DEBUG register to indicate that the core is in sleep mode.

The processor core is woken up from a WEVT instruction when a DMA transfer completes. Execution of code is then resumed from the next instruction in the program sequence. The WEVT instruction uses the completion bits in the DMASTAT1 register as a wake up event. If any completion bit is set to 1, a DMA wake up event is triggered. All completion bits in the DMASTAT1 register must be cleared to disable the event. So, it is necessary to execute a sleep-wake loop to wait until the required DMA channel has completed its data transfer while disabling other completion bits in the process.

Example 10-7 Force a core to enter sleep mode until DMA channel 2 completes its data transfer

```
go_to_sleep:
wevt                ; Execute wait event instruction
lr      r0, [DMASTAT1] ; Core awake, read the completion bits
sr      r0, [DMACENB]  ; An implicit clear of the completion bits
btst    r0, 2          ; DMA channel 2 completed?
beq     go_to_sleep    ; Go back to sleep if DMA channel 2 not completed
```

Pseudo Code

```
WEVT (operand)                                /* WEVT */
    if ((STATUS32.U == 0) || STATUS32.US == 1)
        SLEEP operand
    else
        raiseException (EV_PrivilegeV)
```

Assembly Code Example

The WEVT instruction can be placed anywhere in the instruction sequence.

Example 10-8 WEVT placement in code

```
SUB r2, r2, 0x1
ADD r1, r1, 0x2
WEVT
...
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
WEVT	c	00100000001011110001CCCCC111111
WEVT	<u6>	00100000011011110001uuuuuu111111

XBFU

Function

Extract unsigned bitfield

Extension Group

((SHIFT_OPTION == 2) or (SHIFT_OPTION == 3)) or (BITFIELD_OPTION == 1)

Operation

if (cc) then a = (b >> c[4:0]) & ((1 << (c[9:5]+1)) - 1)

Instruction Format

op b, b, u5, u5

op a, b, c

op a, b, L IMM

op a, L IMM, b

Syntax Example

XBFU<.f> b, b, u5, u5

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Source operand 1 specifies a 32-bit value, from which a bit-field is extracted and assigned to the destination operand. The extracted bit-field starts at bit *N* and includes *M* bits of operand 1.

The value of *N* is specified by the unsigned binary value obtained from the least-significant five bits of operand 2. *M* is obtained by adding 1 to the unsigned five-bit binary value obtained from bits [9:5] of operand 2.

The first source operand is shifted right, without sign-extension, by *N*, and then logically ANDed with a mask of size *M*+1 bits. The result is written into the destination register *a*. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

N = src2[4:0]
M = src2[9:5] + 1
if cc==true then
    dest = (src1 >> N) && ((1 << M)-1)
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
/* XBFU
*first bit to extract
*number of bits to
*extract
*/

```

Assembly Code Example

```

XBFU r1,r1,4,6      ;extract bits 4 through 9 from r1 into r1

XBFU r2,r2,0,5      ; extract bits 0 through 5 from r2 into r2
XBFU r1,r2,r3        ;extract bits from r2 depending on r3
XBFU.F 0,r2,3,5      ;set Z flag if bits 3 to 7 in r2 are zero

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
XBFU<.f>	a,b,c	00100bbb00101101FBBBCCCCCAAAAAA
XBFU<.f>	a,b,u6	00100bbb01101101FBBBuuuuuuAAAAAA
XBFU<.f>	b,b,s12	00100bbb10101101FBBBssssssSSSSSS
XBFU<.cc><.f>	b,b,c	00100bbb11101101FBBBCCCCC0QQQQQ
XBFU<.cc><.f>	b,b,u6	00100bbb11101101FBBBuuuuuu1QQQQQ

If the first source and destination operand are the same register, an XBFU instruction can be encoded using the b, b, s12 operand format, allowing two five-bit constant values to be encoded within the s12 second operand.

If the destination register must be distinct from the first source operand, the two five-bit offsets of the XBFU instruction can be encoded using a LMM value as the second source operand, that is, using format a, b, c where c is a LMM.

An XBFU instruction that extracts a field that is less than or equal to two bits can be encoded using a u6 operand. In this case, only bit 0 of the M operand can be specified, and bits 1 through 4 are assumed to be zero. The N operand can be fully-specified as bits 0 through 4 of the u6 operand. This format allows the source and destination operand registers to be distinct.

op b, b, u6, u6

XOR

Function

Bitwise Exclusive OR

Extension Group

BASELINE

Operation

if (cc) $a = b \wedge c$;

Instruction Format

op a, b, c

Syntax Example

XOR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise exclusive OR of source operand 1 (b) with source operand 2 (c). The result is written into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                /* XOR */
    dest = src1 XOR src2
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
```

Assembly Code Example

```
XOR r1,r2,r3 ; Logical XOR contents of r2 with r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
XOR<.f>	a,b,c	00100bbb00000111FBBBCCCCC(AAAAAA)
XOR<.f>	a,b,u6	00100bbb01000111FBBBuuuuuu(AAAAAA)
XOR<.f>	b,b,s12	00100bbb10000111FBBBssssss(SSSSSS)
XOR<.cc><.f>	b,b,c	00100bbb11000111FBBBCCCCC(0QQQQQ)
XOR<.cc><.f>	b,b,u6	00100bbb11000111FBBBuuuuuu(1QQQQQ)
XOR_S	b,b,c	01111bbbccc00111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

11

The Host

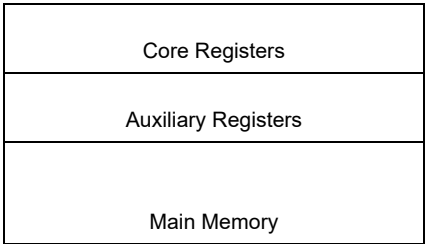
11.1 The Host Interface

The ARCV2-based processor is developed with an integrated host interface to support communications with a host system. The host system can start, stop, or communicate with the ARCV2-based processor using special registers. How the various parts of the ARCV2-based processor appear to the host depends on the host interface.

This section provides an overview of the techniques to control ARCV2-based processor. Most of the techniques are handled by the software debugging system, and the programmer, in general, need not be concerned with these specific details.

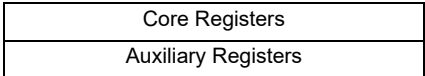
Registers and the program memory of ARCV2-based processor appear as a memory mapped section to the host. [Figure 11-1](#) and [Figure 11-2](#) show an example host system using contiguous part of host memory and an example host system using a section of memory and a section of I/O space, respectively.

Figure 11-1 Example Host Memory Maps, Contiguous Host Memory

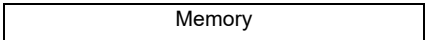


Host I/O Map

Figure 11-2 Example Host Memory Maps, Host Memory and Host I/O



Host Memory Map using a section of the memory



Once a [Reset](#) has occurred, the ARCV2-based processor is put into a known state and executes the initial [Reset](#) code. From this point, the host can make the changes to the appropriate part of the ARCV2-based processor, depending on whether the ARCV2-based processor is running or halted as shown in [Table 11-1](#) on page 780.

Table 11-1 Host Accesses to the ARCV2-based processor

	Running	Halted
Memory	Read/Write	Read/Write
Auxiliary Registers	Read/Write	Read/Write
Core Registers	Read/Write	Read/Write

11.2 Core and Auxiliary Registers

Any debug operation can be carried out while the processor is running. The debug port inserts a pseudo-instruction into the processor pipeline to carry out the requested operation. It is therefore intrusive to normal program execution if carried out while the processor is running (that is, the debug operation steals instruction cycles from the application that is running).

11.3 Memory

The program memory may be changed by the host. The memory can be changed at any time by the host.



Note

If program code is being altered, or transferred into ARCV2-based memory space, the instruction cache must be invalidated.

11.4 Halting

The ARCV2-based processor can halt itself with the FLAG instruction or the host can halt the processor. The host halts the ARCV2-based processor by setting the FH bit in the [Debug Register, DEBUG](#) register.

To force the ARCV2-based processor to halt without overwriting the other status flags, the additional FH bit in the [Debug Register, DEBUG](#) register is provided. The host can test whether the ARCV2-based processor is halted by checking the state of the H bit in the STATUS32 register (see [Status Register, STATUS32](#)). Additionally, the SH bit in the debug register is available to test whether the halt was caused by the host, the ARCV2-based processor, or an external halt signal. The host must wait for the LD (load pending) bit in the [Debug Register, DEBUG](#) register to clear before changing the state of the ARCV2-based processor.

11.5 Starting

The host starts the ARCV2-based processor by clearing the H bit in the [Status Register, STATUS32](#) register.

If the ARCV2-based processor is running code, and needs to be restarted at a different location, you must put the processor into a state similar to the post-[Reset](#) condition to ensure correct operation. The procedure is as follows:

1. Reset the three hardware loop registers to their default values.
2. Disable interrupts, using the status register.
3. Any extension logic must be reset to its default state.
4. Host sets PC to the desired restart location.

If the ARCV2-based processor is running and needs to be restarted to *continue* where the processor left off, the procedure is as follows:

1. The host reads the status from the STATUS Register (see [Status Register, STATUS32](#)).
2. The host writes back to the STATUS32 register (see [Status Register, STATUS32](#)) with *the same* value as was just read, but clearing the H bit.
3. The ARCV2-based processor continues from where it left off when it was stopped.

11.6 Single Instruction Stepping

The Single Instruction Step function is controlled by a bit in the [Debug Register, DEBUG](#) register. The debugger can set this bit to enable Instruction Stepping. The Instruction Step (IS) is write-only by the host and keeps the value for one cycle (see [Table 11-2](#) on page 781).

Table 11-2 Single Step Flags in Debug Register

Field	Description	Access Type
IS	Instruction Step:- Instruction Step enable	Write only from the host

The Single Instruction Step function enables the processor for completion of a whole instruction.

The Single Instruction Step function is enabled by setting the IS bit in the debug register when the processor is halted. The SS bit is ignored.

On the next clock cycle, the processor is kept enabled for as many cycles as required to complete the instruction. Therefore, any stalls because of register conflicts or delayed loads are accounted for when waiting for an instruction to complete. All earlier instructions in the pipeline are flushed, the instruction that the program counter is pointing to is completed, the next instruction is fetched, and the program counter is incremented.

If the stepped instruction is one of the following, two instruction fetches are made, so that the program counter is updated appropriately:

- A Branch, Jump, or Loop with a killed delay slot
- Using Long Immediate data

The processor halts after the instruction is completed.

11.6.1 SLEEP Instruction in Single Instruction Step Mode

The [SLEEP](#) instruction is a NOP instruction (see [Null Instruction Format](#)) when the processor is in the Single Step Mode. Every single-step operation is a restart and the ARCV2-based processor wakes up at the next single-step.

See [“SLEEP”](#) on page [734](#) and [“Null Instruction Format”](#) on page [128](#) for further details.

11.6.2 BRK Instruction in Single Instruction Step Mode

The [BRK](#) instruction behaves exactly as when the processor is not in the Single Step Mode.

11.7 Software Breakpoints

The [BRK](#) instruction can also be used to insert a software breakpoint. [BRK](#) halts the ARCV2-based processor and flushes all subsequent instructions from the processor. The host can then read the PC register to determine where the breakpoint occurred.

Part 2

Security Features

In this part:

- “SecureShield™ Technology”
- “Secure Pipeline”
- “Secure Memory Protection Unit”
- “Secure Debug Access”
- “Error Protection”

12

Security Feature Overview

The core includes the following secure features:

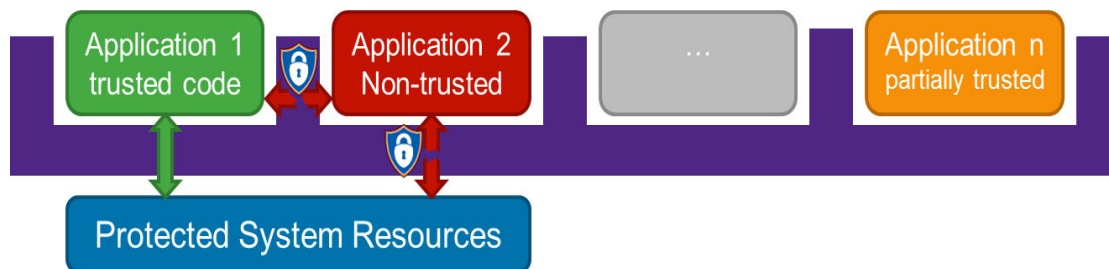
- Securing the Pipeline with ECC—error correction and detection on the register file and program counter.
 - Encrypted Instruction Execution
 - Encrypted Data Access
 - Address and Data Scrambling
- Secure Memory Protection Unit and Secure ID (SID) protection; see [“Secure Memory Protection Unit”](#)
- Configurable secure and normal regions in ICCM0, ICCM1, and DCCM; see [“DCCM”](#) and [“ICCM”](#)
- Programmable secure interrupts and exception handling modes; see [“Interrupts and Exceptions”](#)
- Secure timers; see [“Processor Timers”](#)

13

SecureShield™ Technology

SecureShield™ technology is a set of features supporting execution of applications in isolated, trusted execution environments. Multiple isolated environments can run on a single core. The isolation is mainly enforced by processor privilege levels and an MPU. Depending on the level of trust for the applications running in a specific isolated context, each execution environment can be given or denied access to system resources like memory, memory mapped I/O, CPU control registers, APEX instructions/registers, and so on.

Figure 13-1 SecureShield Overview



13.1 Features of SecureShield™ Technology

Following are the key features of SecureShield™ technology:

- Secure and normal modes for both kernel and user
- Secure MPU
- Secure AHB5 bus signals: HPROT and HNONSEC to indicate normal or secure transfers

13.1.1 SecureShield™ Technology Programming Model

The SecureShield feature supports an easy to use, lightweight, low-overhead application programming model, supporting:

- Set-up, manage protections (memory including stack and peripherals, privileged core registers)
- Provide secure services from one sandbox
- Use secure services from another sandbox

- C function call programming model for using services from other sandbox
- Support for common platform services like interrupt handling, timers, and so on.

13.1.2 SecureShield Protection Concepts

The attributes of the MPU region that is currently executing code determine the security attributes of the ARC SEM core. This permits an implicit secure environment driven by executing code. The addresses of the executing code determine the Secure mode and Secure so that the code determines the trust level. Only the trusted code base can run in the secure processor mode.

13.1.2.1 Operating Privileges

The ARCV2 architecture supports four distinct operating privilege levels: two baseline operating levels: the normal user and kernel mode, secure user and kernel operating levels are provided when the configuration option `-sec_modes_option==true`. These operating privilege levels allow different levels of privilege to be assigned to operating system kernels and user programs. For example, when `-sec_modes_option==true`, you can halt the processor only in the secure kernel mode. Attempts to halt the processor in the normal kernel mode generates a Privilege Violation, except when using the BRK/BRK_S instruction. BRK/BRK_S can halt the processor in the normal kernel mode. The secure user and kernel modes are orthogonal to the normal user and kernel modes.

These operating modes, along with the memory management and protection features, ensure the following:

- An OS can maintain control of the system at all times.
- The OS and user tasks can be protected from a malfunctioning or malicious user task.
- A user task cannot amplify its own privileges.

The operating mode is used to determine whether a privileged instruction may be executed or a privileged register can be accessed. The operating mode is also used by the memory management system to determine whether a specific location in memory may be accessed. The ARCV2 architecture supports the following distinct privilege modes:

- Secure Kernel (available only when `-sec_modes_option==true`)
- Secure User (available only when `-sec_modes_option==true`)
- Kernel (also called as normal kernel mode)
- User (also called as normal kernel mode)

The current mode (secure/normal) of the core is controlled by the MPU region match for the instruction fetch interface. If executing from a region that is marked as secure then the core is in secure mode, if not then it is in normal mode.

Resources (memory regions, auxiliary registers, UAUX & APEX) have access permissions. When a resource is accessed, its access permission is checked against the mode of the instruction attempting the access. This comparison determines if the instruction is allowed to commit or if it generates an exception according to the following table.

13.1.2.2 Secure and Normal Modes

For the C function call programming paradigm to obtain services from other execution contexts with different security privileges, the following are the possible transactions:

1. Normal code calling a secure function (N -> S)
2. A secure function returning to normal mode (S -> N)
3. Secure code calling a normal function (S -> N)
4. A normal function returning to secure mode (N -> S)

Several secure modes are available; see [SecureShield Modes](#).

When normal code calls a secure protection is required to ensure only calls to valid entry points of trusted functions can be done. This is ensured by a secure vector/jump table and corresponding indexed jump instruction. Returning from the secure function to normal mode does not need additional protection, as the trusted code can ensure no secret data is left in registers or other storage before returning.

When secure code calls a normal function, the secure code can ensure a clean context before calling the normal function. However, to ensure the return (case 4) address cannot be tampered with, instead of storing the return address in the BLINK register, the secure code must push it onto a secure stack and pop it on function return.

13.1.2.3 Secure ID

Secure mode switches against due to exceptions and interrupts as well. For protection of leakage of information through registers, normal exceptions/interrupts that occur during secure mode execution is handled after the (secure) registers context is pushed to the secure stack and registers are set to 0.

The Secure ID (SID) provides additional security. The current SID is determined by the MPU region of the current instruction address, and for data accesses the current SID should match the SID of the MPU data region. APEX modules can optionally use SID for the protection of hardware blocks that are not connected through a bus with memory mapped I/O.

13.1.3 SecureShield Modes

A secure mode, orthogonal to kernel and user modes, allows you to run applications in secure isolated environments. For example, you can run an RTOS (with user/kernel modes) in one execution environment and cryptography in another (in secure mode).

The following SecureShield operating modes are available:

- Secure Kernel
- Secure User
- Kernel
- User

13.1.3.1 Secure Mode

The current mode (secure/normal) of the core is controlled by the MPU region match for the instruction fetch interface. If it is executing from a region that is marked as secure, the core is in secure mode; if not, it is in normal mode.

Resources (memory regions, auxiliary registers, UAUX, and APEX) are assigned secure mode access rights. The access rights of the resource, in combination with the mode of the instruction attempting the access, determine if the instruction is allowed to commit, or if it generates an exception according to [Table 13-1](#).

Table 13-1 Core Modes and Resource Permissions

Resource Permission					
		Secure Kernel	Secure User	Normal Kernel	Normal User
Core Mode	Secure Kernel	Yes	Yes	Yes	Yes
	Secure User	No	Yes	No	Yes
	Normal Kernel	No	No	Yes	Yes
	Normal User	No	No	No	Yes

13.1.3.2 Cross-Mode Calling

In C, functions can be defined to the compiler as being secure or normal, and all functions of each type should be grouped together, and kept separate from other sections. These sections should be used in a linker map file to allocate secure and normal code and data to known locations that can then be used to program appropriate regions in the MPU.

Calls can be made from secure to normal mode and normal to secure, however during such calls the user/kernel mode of the core remains unchanged, that, a secure kernel mode task can make calls to normal kernel mode but not directly to normal user mode, and a normal user mode task can make calls to secure user mode but not secure kernel mode.

When switching between secure and normal modes, the hardware automatically swaps between secure and normal versions of various status and configuration registers where they are duplicated for each context.

Switching between Secure and Normal modes (S/N) must always be done through a function call. If code linearly executes across a S/N boundary, jumps or branches without linking from secure to normal, or jumps or branches (with or without linking) from normal to secure, instruction error exceptions are raised on the first instruction/address from the new region.

13.1.4 Register Replication

When `-sec_modes_option==true`, the following registers are duplicated and automatically switched when the core switches between the secure and normal modes.



Note

The replicated registers are internal to the core; these registers are not visible in the auxiliary register space.

- Loop Start & Loop End
- JLI_BASE/LDI_BASE/EI_BASE
- BTA
- Loop count
- Stack pointer registers: AUX_USER_SP, AUX_KERNEL_SP, AUX_SEC_U_SP, and AUX_SEC_K_SP

- Stack checking registers: USTACK_BASE, USTACK_TOP, KSTACK_BASE, KSTACK_TOP, S_USTACK_BASE, S_USTACK_TOP, S_KSTACK_BASE, S_KSTACK_TOP

**Note**

The accumulator registers, if present, must be included in the hardware register stack/zero sequences.

13.1.5 Configuring SecureShield

1. Select **Secure Pipeline Features** in the **Topology View**.

The secure pipeline options appear in the **Options** pane.

2. Select the **2+2 mode option** (`-sec_modes_option`). When this option is set to true, [Table 13-2](#) lists the set of additional instructions and auxiliary registers that are available.

Table 13-2 ARCV2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
<code>-sec_modes_option</code>	false	-	
	true	SJLI, SFLAG	■ NSC_TABLE_TOP ■ NSC_TABLE_BASE ■ AUX_KERNEL_SP ■ AUX_SEC_U_SP ■ AUX_SEC_K_SP ■ AUX_SEC_EXCEPT ■ ERSEC_STAT ■ SEC_STAT ■ SEC_VECBASE_BUILD
<code>- intvbase_preset_s</code>	PC range		INT_VECTOR_BASE_S register. (INT_VECTOR_BASE_S x 1024) Defines the reset value of the programmable INT_VECTOR_BASE_S register. This parameter is also reflected in the ADDR field of the read only SEC_VECBASE_BUILD register. On reset, program execution begins at the address referenced by the reset vector fetched from the 1 KB aligned address (INT_VECTOR_BASE_S x 1024). This option is used for configuring the secure interrupts.

14

Secure Pipeline

Secure pipelining supports the following features on the CCMs and caches:

- Scrambled Instruction Execution
- Scrambled Data Access
- Address and Data Scrambling

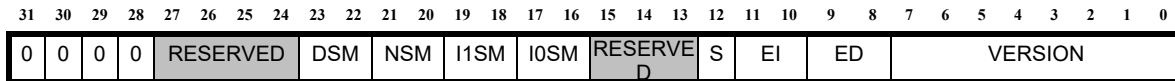
Secure pipelining also supports ECC protection on the register file and program counter. The secure pipeline features are configurable using ARChitect. The corresponding build configuration register is described in this chapter. The ECC protection on the register file and the program counter registers are described in the “[Error Protection](#)” chapter.

14.1 Secure Build Configuration Register, SEC_BUILD

Address: 0xDB

Access: R

Figure 14-1 SEC_BUILD Register



This register indicates the presence of the secure pipeline feature and its configuration, and the secure and normal CCM configurations.

Table 14-1 SEC_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number 0x5: current version. unified branch timing is a sub-feature of unified instruction timing. 0x3: indicates the presence of the following security features: - scrambling and encryption of the cache. - secure pipeline - secure debug - secure MPU - SID protection - key storage protection by MPU - support for the secure kernel and secure user modes - programmable exception handling modes - support for secure interrupts
ED	[9:8]	00: Data encryption is not supported. 01: Data encryption is supported; Read-Modify-Write for sub-word writes is controlled by user encryption logic. All other values are reserved.
EI	[11:10]	00: Instruction encryption is not supported. 01: 16-bit encryption is supported for instructions in the fetch and DA pipeline stages. All other values are reserved.

Table 14-1 SEC_BUILD Field Description

Field	Bit	Description
S	[12]	- Indicates that the core supports secure kernel/user modes and normal kernel/user modes. 0x0: includes only normal kernel and user modes. 0x1: includes secure kernel/user modes and normal kernel/user modes. This bit is valid only when <code>-sec_modes_options==true</code>. Otherwise, this bit is read as zero.
I0SM	[17:16]	ICCM0 secure configuration: 0x0: entire ICCM0 region is normal. 0x1: entire ICCM0 is secure. 0x2: Lower half of ICCM0 is normal; upper-half is secure. 0x3: reserved. - This bit is valid only when <code>-sec_modes_options==true</code>. Otherwise, this bit is read as zero.
I1SM	[19:18]	ICCM1 secure configuration: 0x0: entire ICCM1 region is normal. 0x1: entire ICCM1 is secure region. 0x2: Lower half of ICCM1 is normal; upper-half is secure. 0x3: reserved. - This bit is valid only when <code>-sec_modes_options==true</code>. Otherwise, this bit is read as zero.
NSM	[21:20]	Non-volatile memory secure configuration: 0x0: entire NV_ICCM region is normal. - This bit is always set to 0. This bit is valid only when <code>-sec_modes_options==true</code>. Otherwise, this bit is read as zero.
DSM	[23:22]	ICCM0 secure configuration: 0x0: entire DCCM region is normal. 0x1: entire DCCM is secure region. 0x2: Lower half of DCCM is normal; upper-half is secure. 0x3: reserved. This bit is valid only when <code>-sec_modes_options==true</code>. Otherwise, this bit is read as zero.

15

Secure Memory Protection Unit

ARC SEM provides an optional Memory Protection Option (MPU) which defines up to 16 variable-size memory regions (minimum of 32 bytes and in multiples of 32 bytes thereafter) and assigns read, write, and execute attributes to each of the regions in the secure or normal user and kernel modes. Access violations raise a protection-violation exception.

**Note**

This chapter describes the secure MPU information. This information is valid only when the `-sec_modes_option==true`. When `-sec_modes_option==false`, the behavior of MPU is different: see [Memory Protection Unit](#).

15.1 Protection Overview

The MPU provides the following types of protection:

- Secure memory regions: Define a memory region as secure or normal. Secure memory regions can only be accessed in the secure operating modes. Normal memory regions can be accessed from the secure operating modes. Secure access privileges: Define read, write, and execute operations on a memory regions in the user or kernel mode can also be explicitly programmed.
- SID protection: You can assign secure ID for containers within your software. Where, a container defines a sandbox application with its explicitly managed resources such as private and independent memory area to store private data. You can use the secure MPU to protect these private memories of each container. You can define several containers within your application with different SIDs. To prevent containers from accessing the private memory of other containers, and to avoid programming the MPU each time a container switch occurs, secure MPU provides protection based on SID. Whenever, data has to be fetched, the secure MPU matches the SID of the instruction fetch with the SID of the data fetch. Only if the SIDs match and the secure modes and access privileges of the instruction fetch and data fetch match, only then the data is fetched.

15.2 Configuring an MPU

The ARC EM MPU is an optional component for ARC EM processor cores.

You can add an MPU using the ARChitect GUI. You can then configure the number of regions by changing the GUI parameter or using the following command line:

- `-mpu_num_regions=N`: Specify the number of memory regions as the argument. Acceptable values are 1, 2, 4, 8, and 16. The default is 8.
- `-mpu_sid_option`: Defines if the MPU supports secure region.
 - `False`: SID protection is disabled.
 - `True`: SID protection is enabled.
- `-mpu_prot`: Enable or disable ECC protection on the MPU auxiliary registers.

15.2.1 Enabling the Memory-Protection Unit

The MPU is always enabled.

If the `-mpu_sid_option==true`, the reset value is 0x4001_81C0, that is:

All accesses to the default memory region are treated as secure: read, write, and execute operations are permitted. The permission checks in the MPU region are independent for user and kernel modes. A privilege exception is raised when:

- In kernel mode you try to access resources with user-only permissions
- In user mode you try to access resources with kernel-only permissions

To disable any of the permissions, use the `MPU_EN` register. Also, the SID protection is supported, and the SID for the default region is set to 0x1.

If the `-mpu_sid_option==false`, the reset value is 0x4000_81C0, that is:

All accesses are treated as normal: read, write, and execute operations to the default memory region are permitted in the secure and normal modes. However, SID protection is not supported. The permission checks in the MPU region are independent for user and kernel modes. A privilege exception is raised when:

- In kernel mode you try to access resources with only user-only permissions
- In user mode you try to access resources with only kernel-only permissions

Table 15-1 lists the registers and the instructions that are available in the core when you configure the Secure MPU and SID protection.

Table 15-1 ARCV2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
<code>-mpu_sid_option</code>	false	-	
	true	-	■ Secure MPU registers ■ KEY_STORE_SID

Example 15-1 MPU Kernel and User Mode Independence - Example

```

if (STATUS32.U)
    if (MPU_EN.UR/UW/UE == 0)
        exception raised;
else

```

```
if (MPU_EN.KR/KW/KE == 0)
    exception raised;
```

15.3 Programming MPU Protection

Do the following to program the MPU protection for CCMs in the secure kernel mode:

1. Specify the index of the memory region in the MPU_INDEX register.
2. Specify the start address of the region in the MPU_RSTART register.
3. Specify the end address of the region in the MPU_REND register.



Note

- An MPU region is inclusive of the start and end address.
- The size of the MPU region must be a multiple of 32 bytes.

4. Specify protection attributes for the region:
 - a. Specify if the region is secure or not by programming the MPU_RPER[15] bit.
 - b. Specify the context (SID) for the memory region by programming the MPU_RPER[23:16] field.
 - c. Specify the access attributes (read, write, and execute permissions in the user or kernel mode) by programming the MPU_RPER[8:3] bits.

15.3.1 Programming Considerations

- When a CCM is configured as a completely secure or completely normal region and the CCM size is less than a memory region size (1/16th of the memory), the CCM is repeated in the memory map to occupy the complete region. Define the MPU protection on the CCM to cover the region size (1/16th of the memory) and not just the CCM size. This best practice must be followed because when the CCM size is less than 1/16th of the memory size, the CCM wraps over to be accessible through the whole region; in such scenarios, the region permissions covering the wrap region might differ from those covering the actual CCM address, and region crossing exceptions may be generated if the accesses cross the boundary. Equally, you may get or allow accesses to the memory incorrectly through the wrap regions. Only address of CCMs map to physical location will be used to determine secure or normal attribute defined by Secure CCMs. Wrapping address of CCMs will be used in MPU check to determine S/NS attribute from MPU.
- When a CCM is split into halves of secure and normal memory region, you must define two MPU regions to protect the secure and normal memory regions. Also, the MPU regions must be of the same size as the actual CCM size (not 1/16th of the memory). Also, all the wrap regions require secure mode access. If you define single region to cover the CCM, you must define the whole CCM (including the secure and normal halves) as a secure access in the MPU to allow the entire CCM to be accessed in the secure modes. However, the code and data in the normal half will not be protected from normal mode accesses from the DMI ports. Also, no instructions are allowed to straddle the boundary.
- Overlapping regions are not allowed. Any look up in the MPU (i-fetch or d-fetch) that matches on more than one region causes an EV_MachineCheck exception (ECR: 0x030600). Additionally, the range check is modified to be fully inclusive of the programmed start and end addresses.

- Instructions must be contained completely within one MPU region. Any instruction that crosses an MPU region boundary triggers a region crossing exception.
- For regions defined to cover the instruction memory space, the SID must have only one bit set. When the core does an instruction fetch look up on the MPU, if the SID field is returned with more than one bit set an EV_MachineCheck exception is generated using the ECR 0x031300.

15.4 Software Lookup of the MPU Table

The secure MPU provides a mechanism to look in the MPU table for an address and return the information such as the region index where the looked-up address is located, the start and end address of that MPU region, and the access privileges defined for that address.

To retrieve this information for an address, specify the address in the MPU_PROBE register.

- If any region is hit, the MPU_INDEX register shows the hit region number. The MPU_RSTART/MPU_REND registers show the start/end address of this region. The MPU_RPER register shows the permission configuration for this region (including external pins S input).
- For MPU_RPER aux register, the last data updated is read. It means that, if you write MPU_INDEX, then MPU_RPER will show the permission definition of the region selected by MPU_INDEX (not including external pins S input), if user writes to the MPU_PROBE register, then MPU_RPER is updated to show the permission of this address (including external pins S input).
- The D and M bits are always cleared to zero on SR writes to MPU_INDEX, even if they contain 0x40000000 or 0x80000000.
- Both D and M cannot be set at the same time by MPU_PROBE.
- For writes to the MPU_RSTART, MPU_REND, and MPU_RPER registers, the D and M bits are always ignored, and the I field value indicates the region being written.
- The values returned by reads of the MPU_RSTART, MPU_REND, and MPU_RPER registers after MPU probing are as follows:

Bit Set	MPU_ START and MPU_REND	MPU_RPER
D	0	Permission of the default region
M	0	0
Note that in this table, 0 is simply a value, and does not represent region 0.		

15.5 Protection Mechanism

- For instruction fetch accesses to the ICCMs, NV_ICCM, and IFQ, the operating mode of the instruction is decided by the MPU region definition of the fetch address (whether the region is identified as secure or normal).
 - If an instruction fetch address is in an MPU region marked secure, the core is said to be operating in the secure mode, and the instruction fetch is a secure instruction fetch.
 - If the instruction fetch address is in an MPU region marked normal, the core is said to be operating in the normal mode, and the fetch is performed with normal privilege.

- When the operating mode switches from a normal to a secure mode without any function call or return, the I-fetch invalid S/N transition exception is raised.
- When a function call switches the mode from normal to the secure, no exception is raised.

For information about switching between the operating modes, see [“Switching Between Operating Modes”](#) on page 801.

- For data fetches, the following protections are checked:
 - The operation mode of the instruction is decided by the MPU attributes of the fetched instruction address.
 - the MPU attributes of the data access address is checked against the operation mode of the instruction that requests the data access. When data address is in the secure mode and the instruction is in the normal mode, an *Attempt to access Secure resource from NS mode* exception is raised for this data access.
 - When `-mpu_sid_option==true`, the SID of the instruction fetch and the SID of the data fetch have to match. If the SIDs do not match, an EV_ProtV exception is raised and no memory transaction takes place.

15.5.1 Protection Exception Priority

If a S mode violation and an SID violation occur simultaneously, the S mode violation is reported.

15.5.2 Switching Between Operating Modes

15.5.2.1 Switching Behavior of the Operating Modes

- To allow a function return from a normal mode to the secure mode, following criteria must be met:
 - `SEC_STAT.NSRT == 0`
 - `SEC_STAT.NSRTU==STATUS32.U`
- For an interrupt to return from a normal mode to a secure mode, the following criteria must be met:
 - `SEC_STAT.NSRT == 1`
 - The `SEC_STAT.NSRU==AUX_IRQ_ACT.U` when `AUX_IRQ_ACT[15:0]` indicates the last level of IRQ nesting.
- The return from a secure mode to normal mode from an IRQ is determined by the `SEC_STAT.IRM` bit. The return mode from an exception is determined by the `ERSEC_STAT.ERM` bit. These bits can be programmed in the secure kernel mode. However, in the normal kernel mode, these bits can not be programmed.
- Whether a mode changing micro-operation sequence is pending while a delay slot instruction is executing is determined by the `SEC_STAT.SUE` bit. On exception return, `ERSEC_STAT.SUE` is restored to the `SEC_STAT.SUE` bit and triggers the mode switch micro-op sequence after the delay slot or immediately. You can cancel the pending micro-op sequence before an exception return by clearing the `ERSEC_STAT.SUE` bit.

15.5.2.2 Software-Initiated Switching from Secure to Normal Operating Mode

For calls from secure mode to normal mode, the processor can detect that a branch and link or jump and link is executing in the secure mode and the target of the branch or jump is in a normal memory region, and the processor performs the following steps:

1. Sets the BLINK register to 0x1 to indicate that the call was from secure to normal mode.
2. Saves the return address to the secure stack.
3. Saves the SEC_STAT register to the secure stack.
4. Branches to the target address, automatically switching to the normal mode when fetching from the target address.

For direct software calls to normal region from a secure region, certain internal status is automatically switched from secure to normal mode registers, however no special handling will be included for the register file, so the calling function must take care to leave only the required data present in the register file prior to making the call to prevent information leaking from the secure context. It should be noted that, if this call is due to a compiler inserted support function call then this must be handled in the generated code.

When executing a jump [BLINK] to return from the normal function, a value of 0x1 in the BLINK register causes the processor to pop the real return address from the secure stack before jumping to that address to complete the return.

15.5.2.3 Software-Initiated Switching from Normal to Secure Operating Mode

Switching from a normal to the secure mode is implicitly done when program flow jumps to a instruction in the secure memory. To protect from arbitrary jumps into the secure code, only jumps into controller entry points are allowed. This design is implemented using the SJLI instruction and a secure jump table. The return address in the normal mode is stored in the BLINK register as normal.

If the branches and jumps in the normal mode are linearly executed into the secure address space without using the SJLI instruction, an Instruction Error exception is triggered.

If either the secure jump table or the target of the location in the secure jump table are not located in the secure address space, an EV_MachineCheck exception is triggered when the call is issued.

15.5.2.4 Hardware-Triggered Switching Between Secure and Normal Operating Modes

When a switch from normal mode to secure mode is triggered by hardware (interrupt/exception where the handler is in secure mode) no special handling is needed. However, when the core is in the secure mode and an interrupt or exception is triggered which is to be handled in the normal mode, the secure context is hidden prior to switching to the normal mode context: the full secure context register set is pushed to the secure stack and all the registers in the secure context are set to zero prior to switching to the normal code.

15.6 MPU Auxiliary Register Interface

Table 15-2 MPU Registers

Address	Name	Description
0x6D	"MPU Build Configuration Register, MPU_BUILD"	MPU Build configuration register

Table 15-2 MPU Registers

Address	Name	Description
0x409	"MPU Enable Register, MPU_EN"	MPU enable register
0x448	"MPU Index Register, MPU_INDEX"	MPU index register
0x449	"MPU Region Start Address Register, MPU_RSTART"	MPU region start address
0x44A	"MPU Region End Address Register, MPU_REND"	MPU region end address
0x44B	"MPU Region Permission Register, MPU_RPER"	MPU region permission register
0x44C	"MPU Probe Register, MPU_PROBE"	MPU Probe register

15.6.1 MPU Enable Register, MPU_EN

For information about the MPU_EN register, see [MPU Enable Register, MPU_EN](#).

15.6.1.1 MPU Index Register, MPU_INDEX

Address: 0x448

Access RW in the secure mode only

Figure 15-1 MPU_INDEX for 16 Regions

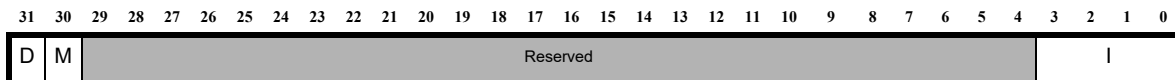
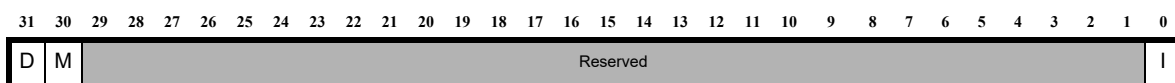


Figure 15-2 MPU_INDEX for 2 Regions



This register is present only when `-sec_modes_option==true`.

Use this register to select an MPU region for programming. When the software looks up an address in the MPU lookup table using the probe register, the MPU_INDEX register returns the index of the MPU region where the lookup address is located. For more information, see [“Software Lookup of the MPU Table”](#) on page 800.

The width of the Index field (`MPU_INDEX[I]`) varies with the number of regions supported. For 2 regions, the index field is a single bit, two bits for four regions, three bits for eight regions, and four bits for 16 regions. If the MPU is configured with a single region, this register is valid but has no useful bits.

Table 15-3 MPU_INDEX Field Description

Field	Description
I	Indicates the region index.
M	Indicates that multiple regions include the lookup address specified in the MPU_PROBE register. This bit is read-only. 0: the lookup address is not present in multiple regions. 1: indicates that multiple regions match the lookup address. The MPU_INDEX[I] and MPU_INDEX[D] bits are set to zero. Also, reads of the MPU_RSTART, MPU_REND, and MPU_RPER registers return zero. This bit retains the value of a probe operation until another probe operation is issued. When software attempts to write to the MPU_INDEX register when this bit is set to 1, the bit is cleared to zero.

Table 15-3 MPU_INDEX Field Description

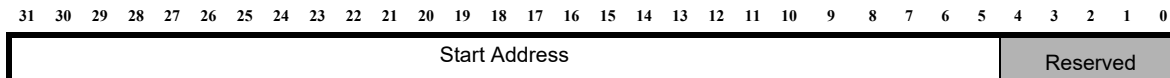
Field	Description
D	Indicates if the lookup address specified in the MPU_PROBE register matches the default region. This bit is read-only. ■ 0: the lookup address is not located in the default region. ■ 1: the lookup address is located in the default region. The MPU_EN register specifies the default region index. The MPU_STARTR and MPU_REND register specify the regions addresses. This bit retains the value of a probe operation until another probe operation is issued. When software attempts to write to the MPU_INDEX register when this bit is set to 1, the bit is cleared to zero.

15.6.1.2 MPU Region Start Address Register, MPU_RSTART

Address: 0x449

Access: RW in the secure mode only

Figure 15-3 MPU_RSTART



This register is present only when `-sec_modes_option==true`.

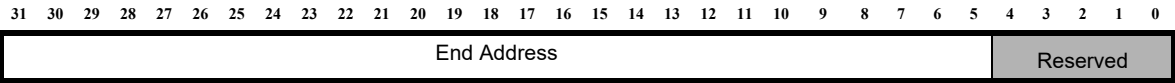
Specifies the start address of the MPU region specified in the MPU_INDEX register. An MPU region is inclusive of the start address. The start address must be a multiple of 32 bytes. Writes to this register set the MPU_RPER[V] bit to zero to prevent creating overlapping MPU regions.

15.6.1.3 MPU Region End Address Register, MPU_REND

Address: 0x44A

Access: RW in the secure mode only

Figure 15-4 MPU_REND



Specifies the end address of the MPU region specified in the MPU_INDEX register. An MPU region is inclusive of the end address. Writes to this register set the MPU_RPER[V] bit to zero to prevent creating overlapping MPU regions. The end address alignment is calculated as follows:

End address = (Multiple of 32 bytes) -1

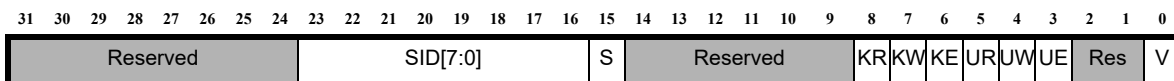
15.6.1.4 MPU Region Permission Register, MPU_RPER

Address: 0x44B

Access: RW in secure mode only

Reset 0x00000000

Figure 15-5 MPU_RPER



This register is present only when `-sec_modes_option==true`.

Use this register to define the MPU protection attributes for an MPU region specified by the MPU_RSTART and MPU_REND registers.

The reset behavior is as follows:

- instruction fetch is not permitted in the specified protection region.
- the specified region is write-protected.
- the specified region is read-protected.
- the specified region is not a secure region.
- the SID of the specified protection region is 0x00.

Table 15-4 MPU_RPER Field Description

Field	Bit	Description
V	[0]	Region valid descriptor ■ 0: region descriptor is unused. ■ 1: region descriptor is valid.
UE	[3]	General Execute Permission ■ 0: Instruction fetch is not permitted in the specified protection region. ■ 1: Instruction fetch is permitted in the specified protection region when in user mode only.
UW	[4]	General Write Permission ■ 0: the specified protection region is write-protected ■ 1: the specified protection region is writable when in user mode only. Note: If MPU_RPER[15]==1, user mode refers to the secure user mode only and kernel mode refers to the secure kernel mode only. If MPU_RPER[15]==0, the user mode can be a normal user mode or the secure user mode.

Table 15-4 MPU_RPER Field Description

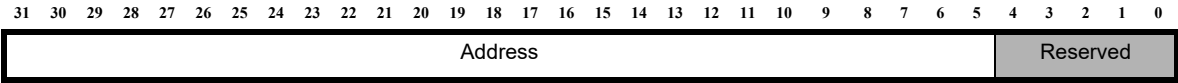
Field	Bit	Description
UR	[5]	General Read Permission 0: the specified protection region is read-protected. 1: the specified protection region is readable when in user mode only. Note that the UR field controls only read access; instruction-fetch access is controlled by the UE field. Note: If MPU_RPER[15]==1, user mode refers to the secure user mode only and kernel mode refers to the secure kernel mode only. If MPU_RPER[15]==0, the user mode can be a normal user mode or the secure user mode
KE	[6]	Kernel-only Execute Permission 0: Instruction fetch is not permitted in the specified protection region 1: Instruction fetch is permitted to the specified protection region when in kernel mode only.
KW	[7]	Kernel-only Write Permission 0: the specified protection region is write-protected 1: the specified protection region is writable when in kernel mode only Note: If MPU_RPER[15]==1, kernel mode refers to the secure kernel mode only. If MPU_RPER[15]==0, kernel mode can be secure kernel mode or the normal kernel mode.)
KR	[8]	Kernel-only Read Permission 0: the specified protection region is read-protected. 1: the specified protection region is readable when in kernel mode only Note that the KR field controls only read access; instruction-fetch access is controlled by the KE field. Note: If MPU_RPER[15]==1, kernel mode refers to the secure kernel mode only. If MPU_RPER[15]==0, kernel mode can be secure kernel mode or the normal kernel mode.)
S	[15]	Indicates whether the specified region is a secure region or normal region. 0: normal region. 1: secure region.
SID[7:0]	[23:16]	Default region SID field. On reset, this field is reset to 0x00. This field is valid only when <code>-sec_modes_option==true</code>. If <code>-sec_modes_option==false</code>, this field is read as zero and ignored on writes. For regions defined to cover instruction memory space, the SID field may only have 1 bit set. When the core does an instruction fetch look up on the MPU, if the SID field is returned with more than one bit set an EV_MachineCheck exception is generated using the ECR 0x031300.

15.6.2 MPU Probe Register, MPU_PROBE

Address: 0x44C

Access: RW in the secure mode only; this register returns zero on reads

Figure 15-6 MPU_PROBE



This register is present only when `-sec_modes_option==true`.

The secure MPU provides a mechanism to lookup the MPU table for an address and return the information such as the region index where the looked up address is located, the start and end address of that MPU region, and the access privileges defined for that address.

For more information, see [“Software Lookup of the MPU Table”](#) on page 800.

Table 15-5 MPU_PROBE Field Description

Field	Description
Address	32-byte aligned address to probe

15.6.3 MPU Build Configuration Register, MPU_BUILD

For information about the MPU_BUILD register, see [Memory Protection Build Configuration Register, MPU_BUILD](#).

15.6.3.1 MPU Exception Cause Register, MPU_ECR

For information about the MPU_ECR register, see [MPU Exception Cause Register, MPU_ECR](#).

16

Secure Debug Access

16.1 Secure Debug Access Overview

When you have configured the processor with secure modes (`-sec_modes_option==true`), you must also ensure that the debug access to the processor matches the operating modes and resource accesses as defined in the processor.

You can configure the debug port to match the processor operating modes, by configuring the secure debug access using the option `-secure_debug==true`. To implement the secure debug access, you must provide two input signals on the debug port: a normal mode unlock signal (`dbg_unlock`) and a secure mode unlock (`dbg_unlock_s`) signal. These signals must indicate the current mode of the debug port.

When secure debug is configured, the debug access port is in one of the following modes:

- **locked mode:** In this mode, the processor is locked, that is, the debugger cannot access the core registers or the memory (excluding the aux registers from 0xF000 to 0xF00F). An attempt to read the core returns the value 0xFFFF_0000 in the IDENTITY register indicating that the core is locked. In this mode, the `dbg_unlock` signal is asserted low. The status of the `dbg_unlock_s` signal does not matter in this mode.
- **normal unlock mode:** In this mode, the instructions inserted by the debugger are treated as operating in the non-secure mode. The debugger can access the non-secure resources in the core such as the non-secure memory regions, and core and auxiliary registers that are accessible in the non-secure mode. Reads to the secure resources are read as zero and writes are ignored. This mode is indicated when the `dbg_unlock` signal is asserted high and the `dbg_unlock_s` signal is asserted low.
- **secure unlock mode:** In this mode, the instructions inserted by the debugger are treated as operating in the secure mode. The debugger can access the secure and non-secure resources in the core such as the non-secure memory regions, secure memory regions, and all core and auxiliary registers. This mode is indicated when the `dbg_unlock` signal is asserted high and the `dbg_unlock_s` signal is asserted high.

16.2 Unlocking a Debug Port

When a debug port is locked, the debugger cannot access the core resources. You must implement your own logic to unlock the debug port. When you have configured an unlock module, you can use the unlock module interface: auxiliary registers 0xF000 to 0xF00F and the unlock module signal interface to unlock the debug port. The debugger can access these registers even when the debug port is locked. A sample

implementation of an unlock module using these auxiliary registers is described in the ARCV2-based processor *Databook*.

16.3 Secure Debug Unlock Module Auxiliary Register Interface

Table 16-1 Secure Debug Registers

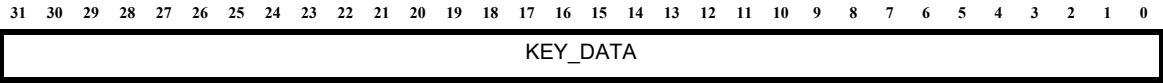
Address	Name	Access
0xF000	"Key Data Register, AUX_KEY_DATA"	Read and write by the debug host only
0xF001	"Key Unlock Register, AUX_KEY_UNLOCK"	Read and write by the debug host only

16.3.1 Key Data Register, AUX_KEY_DATA

Address: 0xF000

Access: RG

Figure 16-1 AUX_KEY_DATA Register



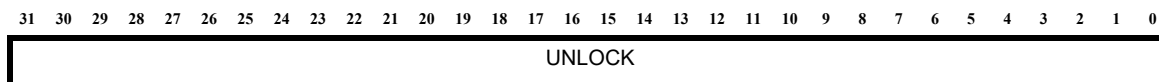
Use this register as part of the debug unlock access to read (challenge) or write (response) the key data. This register is included in the processor only when `sctxdbg_aux_unlk==true`.

16.3.2 Key Unlock Register, AUX_KEY_UNLOCK

Address: 0xF001

Access RG

Figure 16-2 AUX_KEY_UNLOCK Register



Any reads to this register trigger the unlock sequence. Any write to this register indicates that the response key has been completely written to the device and signals the unlock logic to compare the written key with the stored key. If these keys match, the debug channel is unlocked, and the `dbg_unlock` and `dbg_unlock_s` signals are asserted to the debug port. If the keys do not match, the debug port is locked until the next unlock sequence.

This register is included in the processor only when `scdbg_aux_unlk==true`.

17

Error Protection

The ARCV2 ISA provides protection against soft errors in memories caused by external uncontrollable events such as electrical interference and time-based errors using the following modules:

ECC and parity protection

The error protection module provides single-bit error detection and correction and double-bit error detection using ECC and parity algorithms. ECC and parity protection is provided for the following memories:

- CCMs: data and address protection
- Caches: data and tag protection

17.1 Auxiliary Registers

Table 17-1 Error Protection Auxiliary Registers

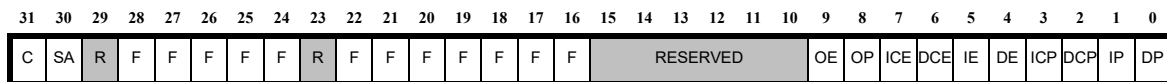
Address	Auxiliary Register Name	Description
0x3F	ERP_CTRL	Error protection control register
0x40	RFERP_STATUS_0	Register File Error Protection Status Register
0x41	RFERP_STATUS_1	Register File Error Protection Status Register
0x405	ESYN	ECC error syndrome register
0xC7	ERP_BUILD	Build Configuration Register: error protection

17.1.1 Error Protection Hardware Control Register, ERP_CTRL

Address: 0x3F

Access: When SecureShield 2+2 mode is not configured: RW
When SecureShield 2+2 mode is configured: RW in the secure mode only

Figure 17-1 ERP_CTRL Register



This control register allows the application to enable or disable the protection hardware built into the processor. Two control bits are assigned for memories. They are used to enable or disable data error detection and correction, and exception generation. This register exists when error protection is configured on the memories. Otherwise, this register is unimplemented. Certain bit fields exist only when the related component is present in the build.

When protection is enabled, the status bits [29:16] in this register indicate the errors that have occurred. These fault bits are sticky, that is, once they are set to 1, they remain set until they are explicitly cleared by writing 0 to the respective bit or writing 1 to bit[31]. When ECC is disabled, the status bits are not updated.



Note

When exceptions are enabled, the instruction fetch address is recorded in the ERET register when a double-bit ECC or Parity error is detected. In addition, the DMP address is recorded in the EFA register when a double-bit ECC or Parity error is detected on load/store accesses.

Table 17-2 ERP_CTRL Field Descriptions

Field	Bit	Reset	Description
DP	[0]	0 if error protection on DCCM is not configured. 1 if error protection on DCCM is configured	Control bit for ECC/parity protection for DCCM; When set to 1, protection is enabled. When cleared to 0, ECC/Parity protection is disabled and no exceptions are raised for ECC, address bus, or parity errors.
IP	[1]	0 if error protection on ICCM0 is not configured. 1 if error protection on ICCM0 is configured	Control bit for ECC/parity protection for ICCM0. When set to 1, protection is enabled. When cleared to 0, ECC/Parity protection is disabled and no exceptions are raised for ECC, address bus, or parity errors.

Table 17-2 ERP_CTRL Field Descriptions (Continued)

Field	Bit	Reset	Description
DCP	[2]	0 if error protection on data cache is not configured. 1 if error protection on data cache is configured	Enable ECC/parity protection for data cache; When set to 1, protection is enabled. When cleared to 0, ECC/Parity protection is disabled and no exceptions are raised for ECC, address bus, or parity errors.
ICP	[3]	0 if error protection on instruction cache is not configured. 1 if error protection on instruction cache is configured	Enable ECC/parity protection for instruction cache; When set to 1, protection is enabled. When cleared to 0, ECC/Parity protection is disabled and no exceptions are raised for ECC, address bus, or parity errors.
DE	[4]	0	Control bit for enabling or disabling protection exceptions for double-bit, address bus, and parity errors on DCCM; A value of 0 enables exceptions, and 1 disables exceptions. On reset, the control bits are set to 0; that is, exceptions are enabled.
IE	[5]	0	Control bit for enabling or disabling protection exceptions for double-bit, address bus, and parity errors on ICCM0; a value of 0 enables exceptions, and 1 disables exceptions. On reset, the control bits are set to 0; that is, exceptions are enabled.
DCE	[6]	0	Control bit for enabling or disabling protection exceptions for double-bit, address bus, and parity errors on data cache; a value of 0 enables exceptions, and 1 disables exceptions. On reset, the control bits are set to 0; that is, exceptions are enabled.
ICE	[7]	0	Control bit for enabling or disabling protection exceptions for double-bit, address bus, and parity errors on instruction cache; A value of 0 enables exceptions, and 1 disables exceptions. On reset, the control bits are set to 0; that is, exceptions are enabled.
OP	[8]	0 if error protection is not configured on overlay manager (OLM) JTLB 1 if error protection is configured on overlay manager (OLM) JTLB	Control bit for ECC/parity protection for OLM JTLB. When set to 1, protection is enabled. When cleared to 0, ECC/Parity protection is disabled and no exceptions are raised for ECC, address bus, or parity errors.
OE	[9]	0	Control bit for enabling or disabling protection exceptions for double-bit, address bus, and parity errors on OLM JTLB; A value of 0 enables exceptions, and 1 disables exceptions. On reset, the control bits are set to 0; that is, exceptions are enabled.
F	[28:16] See Table 17-3	0x000	Double-bit, parity, and address bus faults on CCM and cache accesses when protection is enabled. This field is sticky. To clear a bit, write 0 to that bit or write 0 to ERP_CTRL[31] bit.

Table 17-2 ERP_CTRL Field Descriptions (Continued)

Field	Bit	Reset	Description
R	[29]	0x0	Reserved, read as zero and ignored on writes
SA	[30]	0x0	SBE alarm bit, indicates SBE on a protected target. Writing 1 to this bit clears the status together with the pending SBE interrupt. Writes of zero are ignored.
C	[31]	0x0	Clear ECC/parity interface signals and the ERP_CTRL.F bits that are sticky in functionality. Set bit[31] to 1 to reset the interface signals and the ERP_CTRL.F bits to 0. This bit returns zero when read.

**Note**

- The address bus fault bits in ERP_CTRL are used by the ECC hardware only. Address bus fault bits for the parity hardware are signaled in the same way as a data parity error and therefore occupy bits 16 to 18.
- For details on the error protection operations controlled by the ECC/Parity control bits in the ERP_CTRL register, see the section *Error Protection Actions* in the *DesignWare ARC EM Databook*.

The fault bits in the register in [Table 17-3](#):

Table 17-3 ERP_CTRL Fault Bit Descriptions

Bit Position	Description
16	Double-bit/parity error on DCCM data word
17	Double-bit/parity error on ICCM0 data word
18	Double-bit/parity error on ICCM0 instruction opcode
19	Double-bit/parity error on instruction cache data instruction opcode
20	Double-bit/parity error on data cache data word
21	Double-bit/parity error on instruction cache tag memory
22	Double-bit/parity error on data cache tag memory
24	Address bus fault on DCCM access
25	Address bus fault on ICCM0 access
26	Address bus fault on data cache access
27	Address bus fault on instruction cache access
28	Address bus fault on OLM JTLB memory



Only errors in non-speculative reads from the ARC EM pipeline to the RAMs are recorded; errors in reads from DMA, DMI, and other external system-level components are not recorded.

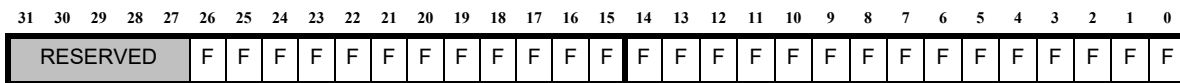
17.1.2 Register File Error Protection Status Register, RFERP_STATUS_0

Address: 0x40

Access:
 ■ When SecureShield 2+2 mode is not configured: RW
 ■ When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00

Figure 17-2 RFERP_STATUS_0 Register



This register indicates whether a single-bit/double-bit/parity error has occurred in a core register. Each bit in this register indicates the fault status of a core register. This register exists when error protection is enabled on the register file or the program counter. Otherwise, this register is unimplemented.

Table 17-4 RFERP_STATUS_0 Field Description

Field	Bit	Description
F	[0]	A value of 1 indicates a single-bit/double-bit/parity error on core register 0.
F	[1]	A value of 1 indicates a single-bit/double-bit/parity error on core register 1.
F	[2]	A value of 1 indicates a single-bit/double-bit/parity error on core register 2.
F	[3]	A value of 1 indicates a single-bit/double-bit/parity error on core register 3.
F	[4]	A value of 1 indicates a single-bit/double-bit/parity error on core register 4.
F	[5]	A value of 1 indicates a single-bit/double-bit/parity error on core register 5.
F	[6]	A value of 1 indicates a single-bit/double-bit/parity error on core register 6.
F	[7]	A value of 1 indicates a single-bit/double-bit/parity error on core register 7.
F	[8]	A value of 1 indicates a single-bit/double-bit/parity error on core register 8.

Table 17-4 RFERP_STATUS_0 Field Description

Field	Bit	Description
F	[9]	A value of 1 indicates a single-bit/double-bit/parity error on core register 9.
F	[10]	A value of 1 indicates a single-bit/double-bit/parity error on core register 10.
F	[11]	A value of 1 indicates a single-bit/double-bit/parity error on core register 11.
F	[12]	A value of 1 indicates a single-bit/double-bit/parity error on core register 12.
F	[13]	A value of 1 indicates a single-bit/double-bit/parity error on core register 13.
F	[14]	A value of 1 indicates a single-bit/double-bit/parity error on core register 14.
F	[15]	A value of 1 indicates a single-bit/double-bit/parity error on core register 15.
F	[16]	A value of 1 indicates a single-bit/double-bit/parity error on core register 16.
F	[17]	A value of 1 indicates a single-bit/double-bit/parity error on core register 17.
F	[18]	A value of 1 indicates a single-bit/double-bit/parity error on core register 18.
F	[19]	A value of 1 indicates a single-bit/double-bit/parity error on core register 19.
F	[20]	A value of 1 indicates a single-bit/double-bit/parity error on core register 20.
F	[21]	A value of 1 indicates a single-bit/double-bit/parity error on core register 21.
F	[22]	A value of 1 indicates a single-bit/double-bit/parity error on core register 22.
F	[23]	A value of 1 indicates a single-bit/double-bit/parity error on core register 23.
F	[24]	A value of 1 indicates a single-bit/double-bit/parity error on core register 24.
F	[25]	A value of 1 indicates a single-bit/double-bit/parity error on core register 25.
F	[26]	A value of 1 indicates a single-bit/double-bit/parity error on core register 30.

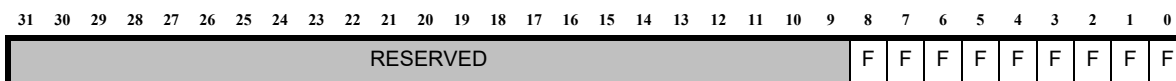
17.1.3 Register File Error Protection Status Register, RFERP_STATUS_1

Address: 0x41

Access:
 ■ When SecureShield 2+2 mode is not configured: RW
 ■ When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00

Figure 17-3 RFERP_STATUS_1 Register



This register indicates whether a single-bit/double-bit/parity error has occurred in a core register. Each bit in this register indicates the fault status of a core register. This register exists when error protection is enabled on the register file or the program counter. Otherwise, this register is unimplemented.

Table 17-5 RFERP_STATUS_1 Field Description

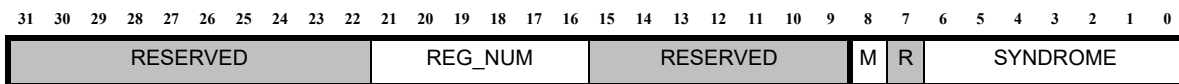
Field	Bit	Description
F	[0]	A value of 1 indicates a single-bit/double-bit/parity error on ACCL.
F	[1]	A value of 1 indicates a single-bit/double-bit/parity error on ACCH.
F	[2]	A value of 1 indicates a single-bit/double-bit/parity error on GP.
F	[3]	A value of 1 indicates a single-bit/double-bit/parity error on FP.
F	[4]	A value of 1 indicates a single-bit/double-bit/parity error on SP.
F	[5]	A value of 1 indicates a single-bit/double-bit/parity error on ILINK.
F	[6]	A value of 1 indicates a single-bit/double-bit/parity error on BLINK.
F	[7]	A value of 1 indicates a single-bit/double-bit/parity error on LP_COUNT.
F	[8]	A value of 1 indicates a single-bit/double-bit/parity error on PC.

17.1.4 ECC Syndrome Register, ESYN

Address: 0x405

- Access:
- When SecureShield 2+2 mode is not configured: RW
 - When SecureShield 2+2 mode is configured: RW in the secure mode only

Figure 17-4 ESYN Register



This register is present only if the processor is configured with error protection. When ECC exceptions are raised, you can read this register to determine the register that has triggered the exception and the kind of ECC error that has triggered the exception.

When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV_Privilege violation exception. On exception entry, the bit is cleared. When an exception is triggered by an instruction in the non-secure mode, the internal bit is cleared to 0.



Note

When any related exception is taken, all the fields in the ESYN register are updated. The fields unrelated to the triggered exception will have undefined value.

Table 17-6 ESYN Field Description

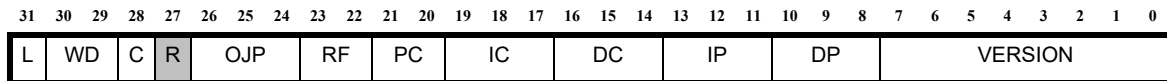
Field	Bit	Description
SYNDROME	[6:0]	Indicates the ECC syndrome for CCM and cache.
M	[8]	If this bit is set to 1: indicates that multiple ECC errors have occurred during a copy-back operation. This bit is present only for the processors with data cache. If data cache is not present in a processor, this bit is read as zero and ignored on writes.
REG_NUM	[21:16]	Indicates the register number that has triggered the register file ECC exception. If ECC protection is not configured on the processor register file, this bit is read as zero and ignored on writes. If PC protection is also configured, and a PC exception is triggered and without any RF error, this field is updated with the value 63.

17.1.5 Error Protection Configuration Register, ERP_BUILD

Address: 0xC7

Access: R

Figure 17-5 ERP_BUILD Register



The ERP_BUILD register contains information about the safety features included in the processor. This register is a read-only register and reflects the options selected when building the processor in the ARChitect configuration tool. This register exists when any of the following features are configured: lockstep interface, watchdog timer, or error protection on instruction cache, data cache, ICCM0, and DCCM. Otherwise, this register is unimplemented.

Note that the fields [27:14] are reserved in the ARC EM4 processor series.

[Table 17-7](#) lists the field descriptions.

Table 17-7 ERP_BUILD Register

Field	Bit	Description
VERSION	[7:0]	Indicates the error protection version. 0x6: Added support for detecting all zeros and ones on data. 0x8: Includes support for ECC protection on the ICCM0 partition and OLM JTLB memories.
DP	[10:8]	Indicates the presence of DCCM protection and its configuration. 0x0: None 0x1: ECC protection on data 0x2: ECC protection on data and DCCM address 0x3: Parity protection on data 0x4: Parity protection on data and address
IP	[13:11]	Indicates the presence of ICCM0 protection and its configuration. 0x0: None 0x1: ECC protection on data 0x2: ECC protection on data and DCCM address 0x3: Parity protection on data 0x4: Parity protection on data and address

Table 17-7 ERP_BUILD Register

Field	Bit	Description
DC	[16:14]	Indicates presence of data cache protection and its configuration. 0x0: None 0x1: ECC protection on data and tag 0x2: ECC protection on data, tag, and index 0x3: Parity protection on data and tag 0x4: Parity protection on data, tag, and index
IC	[19:17]	Indicates presence of instruction cache protection and its configuration. 0x0: None 0x1: ECC protection on data and tag 0x2: ECC protection on data, tag, and index 0x3: Parity protection on data and tag 0x4: Parity protection on data, tag, and index
PC	[21:20]	Indicates presence of program counter protection and its configuration. 0x0: None 0x1: ECC protection on the program counter
RF	[23:22]	Indicates presence of the register file protection and its configuration. 0x0: None 0x1: ECC protection the register file
OJP	[26:24]	Indicates presence of the OLM JTLB memory protection and its configuration. 0x0: None 0x1: ECC protection on data and address
APX	[27]	Indicates whether the secure apex is protected by ECC. 0x0: no protection 0x1: ECC protection
C	[28]	Indicates whether errors can be corrected on-the-fly, that is, correct the error data when the data is fetched from CCM. The error data is replaced with the corrected data and there will be no correction write-back to the CCM. 0x0: errors are corrected and written to the CCMs 0x1: errors are corrected but are not written to the CCMs

Table 17-7 ERP_BUILD Register

Field	Bit	Description
WD	[30:29]	Indicates the presence of the watchdog timer and its configuration. ■ 0x0: None ■ 0x1: 16-bit timer ■ 0x2: 32-bit timer Note: When the windowed watchdog component is configured, the ERP_BUILD.WD field reflects the counter width of the first watchdog timer. The counter width of the other watchdog timers can be determined from the EWDT_BUILD register.
L	[31]	Indicates presence of the lockstep interface. ■ 0x0: lockstep interface is not available ■ 0x1: lockstep interface is available

17.2 Exceptions

See [Table 6-6](#) on page 268.

Part 3

Memory and System Components

In this part:

- “Memory Protection Unit”
- “Protection Schemes”
- “ICCM”
- “Instruction Fetch Queue”
- “Instruction Cache”
- “DCCM”
- “Data Cache”
- “Task Switching Component”
- “Overlay Management Unit”
- “Program Storage”
- “Calibration Parameter Storage”
- “Cryptographic Key Storage”
- “Processor Timers”
- “Enhanced Watchdog Timer”
- “Peripheral Bus”
- “External Host Debugging”

- “Debug Features”
- “Performance Counters”
- “ARC Trace”
- “Power Management Features”
- “Floating-Point Unit”
- “XY”
- “Address Generation Unit Auxiliary Interface”
- “Bitstream Processing”
- “User Auxiliary Register Interface”
- “ARConnect”
- “micro DMA Controller”

18

Memory Protection Unit



Note

This chapter describes the MPU information when the SecureShield 2+2 (-sec_modes_option==false) modes and the Secure MPU are not configured. When your processor includes the SecureShield 2+2 modes (-sec_modes_option==true) and the Secure MPU, the behavior is different. See [Secure Memory Protection Unit](#) for more information.

18.1 Memory Protection Unit

The ARCV2-based Memory Protection Unit (MPU) provides protection by dividing the address space into regions associated with specific attributes such as Read, Write, and Execute. If an attempt is made to access a region for which an associated attribute is not permitted, the ARCV2-based processor raises a Protection Violation exception, and this exception prevents the faulting instruction from completing.

The ARCV2-based processor provides kernel-mode capabilities such as separate Read, Write, and Execute permissions for the user and kernel mode processes on each memory region.

The provision of distinct kernel and user permissions allows an operating system to protect its code and data from illegal or unexpected accesses by user-mode processes. A kernel-mode process is always permitted to access any region for which the required permission is provided to user-mode processes.

You can define default kernel and user permissions for accesses that fall outside all memory protection regions.

Multiple MPU regions are allowed to overlap; the attributes that apply to a particular address in the case of overlapping regions are determined using a fixed priority scheme.

18.1.1 Key Features

The following are the main features of the ARCV2-based processor MPU:

- Programmable execute permission bits to enable or disable execution of code from specific regions of memory
- Programmable data read and write permission bits to enable or disable data access to specific regions of memory
- Separate kernel and user mode read, write, and execute permissions
- 1, 2, 4, 8 or 16 configurable memory regions.
- Regions that can be programmed individually and independently
- Overlapping regions supported by a priority scheme
- Programmable default permissions that apply to accesses outside all programmed protection regions
- EV_ProtV (see [Exception Vectors and the Exception Cause Register](#) on page 263) exception to indicate access violations
- Protection exceptions that are precise and can be restarted
- Designed to be used in conjunction with the Stack Checking and/or Code Protection mechanisms in the ARCV2-based processors
- Non-cacheable regions can be specified using the orthogonal features provided by the [Non-cached Memory Region, AUX_CACHE_LIMIT](#) register in the ARCV2-processor.

18.1.2 Configuration Options

The ARCV2 ISA MPU is an optional component for the ARCV2-based processors. The configuration option `mpu_num_regions` defines the number of regions supported by the MPU.

18.1.3 Enabling the MPU

All MPU registers contain all zeros after reset. To enable the MPU, set the MPU_EN register to 0x40000000. For more information about the register, see [MPU Enable Register, MPU_EN](#).

18.1.4 Data Organization and Addressing

The ARCV2 address space is divided into 16 equal-sized regions that are used to define the locations of memories (ICCM and DCCM), as well as the location of peripherals and uncacheable memory.

The ARCV2 MPU allows software to further define a number of variable-sized protection regions with programmable attributes that control what types of reference are permitted in each protection region. The maximum number of protection regions is fixed at configuration time and is specified in the MPU_BUILD register ([Memory Protection Build Configuration Register, MPU_BUILD](#)).

Protection regions are allowed to overlap each other: the attributes that apply to a particular address in the case of overlapping regions are determined using a fixed priority scheme. This mechanism can be used to implement a flexible organization based on need, for example, a large read-only region with a smaller embedded writable region.

Each region has a base address and size, specified in the [MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB15](#) and [MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP15](#). The base address of each region must be aligned to a multiple of its region size, which is always a power of two.

When an access to a protection region raises a violation, the access is not permitted to succeed and a Protection Violation exception is raised. The [MPU Exception Cause Register, MPU_ECR](#) indicates the type of transaction that causes the violating access, and the MPU Exception Cause Register (MPU_ECR) provides additional information about the faulting access.

All reserved register bits must be written as 0 to ensure correct operation, and all reserved fields read back as 0.

To enable execution of code to set up the MPU shortly after reset, the MPU comes out of reset in a disabled state with MPU_EN and all its region-descriptor registers set to 0x00000000.



Note

The MPU protects the complete memory space on the processor. When you update the region descriptors, ensure that both the interrupt vector space and the code modifying the region-descriptors are executable, even during the update. You must disable the MPU before any region descriptors are updated by setting the MPU_EN register to 0x00000000. After you set all the region descriptors appropriately, you can re-enable MPU by setting MPU_EN to 0x40000000.

The following examples show different memory maps features and corresponding register set up:

18.1.4.1 Example 1 - Contiguous Regions

In this example, the memory has the following features:

- A user-executable region in the first half of memory
- A user-readable and user-writable region in the next quarter of memory
- A user-readable and user-writable region for peripherals in the last quarter. This region coincides with the uncached region programmed into the AUX_CACHE_LIMIT register with the value 0xC0000000

Example 1 Memory Map shows the memory map for this set up:

Table 18-1 Example 1 -- Contiguous Regions Memory Map

Region 0	0x00000000	Base address: 0x00000000
	0x7FFFFFFF	Region size: 2G
Region 1	0x80000000	Regions permissions: user-execute
	0xBFFFFFFF	Base address: 0x80000000
Region 2	0xC0000000	Region size: 1G
	0xFFFFFFF	Regions permissions: user-read, user-write
	0xC0000000	Base address: 0xC0000000
	0xFFFFFFF	Region size: 1G
	0xFFFFFFF	Regions permissions: user-read, user-write

The base address and size, as specified in the MPU Region Descriptor Base Registers and MPU Region Descriptor Permissions Registers, is set up as shown in the following figures:

Figure 18-1 Example 1: MPU_RDB0 Register

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	
BASE_ADDRESS																												RESERVED				V
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Figure 18-2 Example 1: MPU_RDP0 Register

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
RESERVED																				SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	1	0

Figure 18-3 Example 1: MPU_RDB1 Register

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	
BASE_ADDRESS																												RESERVED				V
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Figure 18-4 Example 1: MPU_RDP1 Register

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
RESERVED																				SIZE[4:2]				KR	KW	KE	UR	UW	UE	R	SIZE
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1

Figure 18-5 Example 1: MPU_RDB2 Register

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	
BASE_ADDRESS																												RESERVED				V
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Figure 18-6 Example 1: MPU_RDP2 Register

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
RESERVED																				SIZE[4:2]				KR	KW	KE	UR	UW	UE	R	SIZE
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1

18.1.4.2 Example 2 - Overlapping Regions

In this example the memory has the following features:

- The entire memory space is defined as user-readable and user-writable using the lowest priority region (for an 8-region setup this region is region 7).
- There is then an overlapping user-accessible, execute-only region in the lower 1G of memory to contain the vector table and kernel operating system code.

The memory map for this setup is shown in the following figure:

Figure 18-7 Overlapping Regions Memory Map

Region 0 overlaps with Region 7	0x00000000	Base address: 0x00000000 Region size: 1G Region permissions: user- execute	
	0x3FFFFFFF		
Region 7	0x40000000		Base address: 0x00000000 Region size: 4G Region permissions: user-read, user-write
	0xFFFFFFFF		

The base address and size, as specified in the MPU Region Descriptor Base Registers and MPU Region Descriptor Permissions Registers, are set up as shown in the following figures:

Figure 18-8 Example 2: MPU_RDB0 Register

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	
BASE_ADDRESS																												RESERVED				V
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		

Figure 18-9 Example 2: MPU_RDP0 Register

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
RESERVED																				SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	0	1

Figure 18-10 Example 2: MPU_RDB7 Register

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	
BASE_ADDRESS																												RESERVED				V
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		

Figure 18-11 Example 2: MPU_RDP7 Register

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
RESERVED																				SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	1	1

The base address and size, as specified in the MPU Region Descriptor Base Registers and MPU Region Descriptor Permissions Registers, is setup as shown in the following figures:

Figure 18-12 Example 1: MPU_RDB0 Register

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
BASE_ADDRESS																					RESERVED							V			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 18-13 Example 1: MPU_RDP0 Register

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
RESERVED																				SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	1	0

Figure 18-14 Example 1: MPU_RDB1 Register

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
BASE_ADDRESS																					RESERVED							V			
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Figure 18-15 Example 1: MPU_RDP1 Register

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
RESERVED																				SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1

Figure 18-16 Example 1: MPU_RDB2 Register

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
BASE_ADDRESS																					RESERVED								V		
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Figure 18-17 Example 1: MPU_RDP2 Register

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
RESERVED																				SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1

18.1.4.3 Example 2 - Overlapping Regions

In this example the memory has the following features:

- The entire memory space is defined as user-readable and user-writable using the lowest priority region (for an 8 region setup this region is region 7).
- There is then an overlapping user-accessible, execute-only region in the lower 1G of memory to contain the vector table and kernel operating system code.

The memory map for this set up is shown in the following figure:

Figure 18-18 Overlapping Regions Memory Map

Region 0 overlaps with Region 7	0x00000000	Base address: 0x00000000 Region size: 1G Region permissions: user-execute	
	0x3FFFFFFF		
Region 7	0x40000000		Base address: 0x00000000 Region size: 4G Region permissions: user-read, user-write
	0xFFFFFFFF		

The base address and size, as specified in the MPU Region Descriptor Base Registers and MPU Region Descriptor Permissions Registers, is set up as shown in the following figures:

Figure 18-19 Example 2: MPU_RDB0 Register

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
BASE_ADDRESS																					RESERVED										V
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 18-20 Example 2: MPU_RDP0 Register

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
RESERVED																				SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	0	1

Figure 18-21 Example 2: MPU_RDB7 Register

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
BASE_ADDRESS																					RESERVED							V			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 18-22 Example 2: MPU_RDP7 Register

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
RESERVED																				SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	1	1

18.1.5 Modes of Operation

After reset, the ARCV2 registers contain default values that disable the ARCV2-based processor. All the region descriptor registers contain zero.

The MPU control and status registers are described in section Register Set Details.

Different memory organization and addressing configuration examples are shown in section [Data Organization and Addressing](#).

The following sections cover the modes of operation in more detail:

- [Region Accesses](#)
- [Memory Protection Exceptions](#)
- [Multiple Simultaneous Protection Violations](#)

18.1.5.1 Region Accesses

All accesses are checked against the region descriptors, according to a fixed priority, to determine what attributes to apply.

Table 18-2 Priority Configurations

Region	Priority (8-region configuration)	Priority (16-region configuration)
0	Highest	Highest
1	2 nd Highest	2 nd Highest
2	3 rd Highest	3 rd Highest
3	4 th Highest	4 th Highest
4	5 th Highest	5 th Highest
5	6 th Highest	6 th Highest
6	7 th Highest	7 th Highest
7	Lowest	8 th Highest
8	Not applicable	9 th Highest
9	Not applicable	10 th Highest
10	Not applicable	11 th Highest
11	Not applicable	12 th Highest
12	Not applicable	13 th Highest
13	Not applicable	14 th Highest
14	Not applicable	15 th Highest
15	Not applicable	Lowest

If an address is covered by more than one region, the attributes that relate to the highest priority region apply, and the contents of the MPU_ECR register reflect this priority when an exception is taken.

To protect only a small region of the memory map, the lowest priority region must be set to allow all access types. Higher priority regions can then be set to protect progressively smaller regions as desired.

If an address is covered by none of the enabled protection regions, the default permissions from the [MPU Enable Register, MPU_EN](#) register are applied.

18.1.5.2 Memory Protection Exceptions

When the MPU detects a memory protection violation, the MPU raises the EV_ProtV exception. If there is no higher priority exception outstanding, the EV_ProtV exception is taken. As a result, the faulting instruction is discarded and an exception occurs. If the faulting instruction was a Store instruction, the store to memory operation does not occur. A faulting instruction fetch or load instruction may have taken place, but the memory is not modified, and the normal register updates defined by the faulting instruction do not occur.

[Table 18-3](#) lists the settings of the ECR register based on the entry to the EV_ProtV exception.

Table 18-3 Setting ECR Register based on EV_ProtV Exception

Failed Reference type	Vector Name	Vector Offset	Vector Number	Cause Code	Parameter	ECR value
Memory Read (such as LD, POP, LEAVE, interrupt exit, LLOCK)	EV_ProtV	0x18	0x06	0x01	0x04	0x060104
Memory Write (such as ST, PUSH, ENTER, interrupt entry, SCOND)	EV_ProtV	0x18	0x06	0x02	0x04	0x060204
Memory Read-Modify-Write (such as EX)	EV_ProtV	0x18	0x06	0x03	0x04	0x060304

Bit 2 of the 8-bit parameter field is set to 1 to indicate that a protection violation was reported by the MPU.

The exception vector table and the EV_ProtV exception handler must always be in a valid region with at least kernel execute permission. If they are not, the processor raises a double exception on accessing the exception vector table or the handler and triggers a non-restartable Machine Check exception.

When execution enters the exception handler, the address that caused the protection violation is available in the [Exception Fault Address, EFA](#) (0x404).

In the case of instruction-fetch violations, the EFA register is set to the address of the instruction fetch that caused the violation. Care must be taken to ensure that the instruction or data elements do not straddle MPU region boundaries as access protection checks are performed only using the starting address (first byte at the lowest address) of the element. In the case of data access violations, EFA is set to the address of the data access that caused the violation.



Note

When arranging MPU regions and user code, care must be taken to ensure that when a branch has a delay slot, both the branch and its delay slot always reside within a single MPU region or in regions that have the same set of access permissions.

18.1.5.3 Multiple Simultaneous Protection Violations

In a processor with a precise exception model, the first violation in program order is the one that must be taken. This model requires that all outstanding instructions that have already been committed must complete and all uncommitted instructions must be discarded. Faulting instructions may then be retried after the violation had been dealt with.

In ARCV2-based processors, all protection violation exceptions are precise. Therefore, all non-violating instructions that are issued before a protection violation are detected and allowed to complete, and the instruction that raises the protection violation is dismissed. Any instructions that occur after the faulting instruction, and which themselves may cause protection violations, are also dismissed when the faulting instruction is dismissed.

In the case of simultaneous instruction and data violations generated by a single instruction, the instruction fetch violation is reported and the data violation is ignored as ARCV2 architecture gives higher priority to instruction fetch violations.

Due to the pipelined construction of the ARCV2-based processor, the instruction fetch and data access violations are detected by hardware at the same time, even though the data violation arises from an earlier instruction than the instruction fetch violation. In this case, the data violation takes precedence, as the violation occurs earlier in program order.

Following are the potential sources of a protection violation in the ARCV2 architecture: Code Protection, Stack Checking, MPU, and MMU. The MPU and MMU options are mutually exclusive, and therefore exceptions from these two sources cannot occur simultaneously. However, you can configure an ARCV2-based processor with MPU, Stack Checking, and Code Protection, and so one Load or Store instruction to violate the protection policies of all three schemes simultaneously. As Stack Checking and Code Protection policies apply only to Load or Store types of references, only the MPU (or MMU) can raise protection violations in relation to instruction fetches.

Each source of protection violation independently sets a different bit in the parameter field of the Exception Cause Register (ECR), allowing multiple simultaneous violations to be reported by the same exception. For reference they are summarized in the following table:

Table 18-4 ECR parameter values for EV_ProtV sources

Source of EV_ProtV	ECR Parameter Value
Code protection	0x01
Stack Checking	0x02
MPU	0x04

Therefore, if the address of a POP instruction is detected by the MPU as a protection violation, as well as having an address that is outside the defined stack region, and also reads from a code area that is protected by the Code Protection mechanism, the ECR parameter value is 0x7 (MPU, Stack Checking, and Code Protection bits all set).

However, if the same instruction also raises a protection violation due to its PC value being within a region that does not have execute permission, the instruction violation takes precedence over all data violations from whatever source, thus the ECR parameter value is 0x4 (MPU only).

18.1.5.4 Caution When Updating Region Settings

As the MPU protects all the memory space on an ARCV2-based processor, care must be taken when updating its region descriptors to ensure that both the exception vector space and the code modifying the region descriptors remain executable, even during the update. Therefore, you must disable the MPU before any region descriptors are updated by setting the MPU_EN register to 0x00000000. After all the region descriptors have been set up appropriately, the MPU can be re-enabled by setting MPU_EN to 0x40000000. The instruction immediately following an AUX write enabling the MPU should not reside in a no-execute MPU region.

Similarly, when you modify the region descriptors through the debugger via host access when the CPU is running, you must disable the MPU before any region descriptors are updated by setting the MPU_EN register to 0x00000000.

The debugger is never prevented by the MPU from accessing memory at any position in the memory map.

If a region descriptor is programmed with reserved values in the size field, the behavior of that region is undefined if the region and the MPU are both enabled. In practice, this kind of programming may lead to unexpected EV_ProtV violations being reported, or to the absence of such reports from any such incorrectly programmed MPU region. Therefore, use only legal values in the region descriptors when they are enabled. A disabled region descriptor never triggers a protection violation and can be programmed with illegal values without causing problems.

18.1.6 Memory Protection Unit Registers

ARCV2-based processors support an optional Memory Protection Unit (MPU) that contains a collection of control registers to define the behavior of the MPU and status registers to indicate the source of a memory protection violation. [Table 18-5](#) lists the MPU auxiliary registers.

Table 18-5 MPU Register Set

Address	Auxiliary Register Name	Description
0x6D	MPU_BUILD	MPU build configuration register
0x409	MPU_EN	MPU enable and default permission register
0x420	MPU_ECR	MPU exception cause register
0x422	MPU_RDB0	MPU region descriptor base 0
0x423	MPU_RDP0	MPU region descriptor permissions 0
0x424	MPU_RDB1	MPU region descriptor base 1
0x425	MPU_RDP1	MPU region descriptor permissions 1
0x426	MPU_RDB2	MPU region descriptor base 2
0x427	MPU_RDP2	MPU region descriptor permissions 2
0x428	MPU_RDB3	MPU region descriptor base 3
0x429	MPU_RDP3	MPU region descriptor permissions 3

Table 18-5 MPU Register Set

Address	Auxiliary Register Name	Description
0x42a	MPU_RDB4	MPU region descriptor base 4
0x42b	MPU_RDP4	MPU region descriptor permissions 4
0x42c	MPU_RDB5	MPU region descriptor base 5
0x42d	MPU_RDP5	MPU region descriptor permissions 5
0x42e	MPU_RDB6	MPU region descriptor base 6
0x42f	MPU_RDP6	MPU region descriptor permissions 6
0x430	MPU_RDB7	MPU region descriptor base 7
0x431	MPU_RDP7	MPU region descriptor permissions 7
0x432	MPU_RDB8	MPU region descriptor base 8
0x433	MPU_RDP8	MPU region descriptor permissions 8
0x434	MPU_RDB9	MPU region descriptor base 9
0x435	MPU_RDP9	MPU region descriptor permissions 9
0x436	MPU_RDB10	MPU region descriptor base 10
0x437	MPU_RDP10	MPU region descriptor permissions 10
0x438	MPU_RDB11	MPU region descriptor base 11
0x439	MPU_RDP11	MPU region descriptor permissions 11
0x43a	MPU_RDB12	MPU region descriptor base 12
0x43b	MPU_RDP12	MPU region descriptor permissions 12
0x43c	MPU_RDB13	MPU region descriptor base 13
0x43d	MPU_RDP13	MPU region descriptor permissions 13
0x43e	MPU_RDB14	MPU region descriptor base 14
0x43f	MPU_RDP14	MPU region descriptor permissions 14
0x440	MPU_RDB15	MPU region descriptor base 15
0x441	MPU_RDP15	MPU region descriptor permissions 15

18.1.7 MPU Enable Register, MPU_EN

Address: 0x409

- Access:
- When SecureShield 2+2 mode is not configured: RW
 - When SecureShield 2+2 mode is configured: RW in the secure mode only

The MPU_EN register enables or disables all MPU functionality and defines the default permissions that apply to accesses falling outside of all protection regions.

Memory that is not covered by any enabled protection region descriptor is referred to as part of the default protection region.

When secure MPU is not configured:

This register contains 0x00000000 on reset.

When secure MPU is configured:

For a build without the `-mpu_sid_option`, the reset value is 0x4000_81C0 and for a build with `-mpu_sid_option==true`, the reset value is 0x4001_81C0.

Figure 18-23 MPU_EN Register

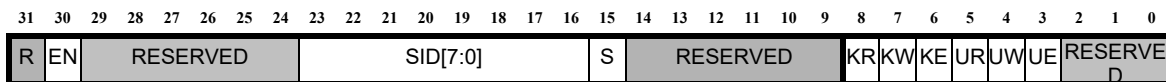


Table 18-6 MPU Enable Register

Field	Bit	Description
UE	[3]	General Execute Permission ■ 0: Instruction fetch is not permitted in the default protection region in user mode. ■ 1: Instruction fetch is permitted in the default protection region in user or kernel mode (irrespective of the KE bit value).

Table 18-6 MPU Enable Register

Field	Bit	Description
UW	[4]	General Write Permission 0: Default protection region is write-protected in user mode. 1: Default protection region is writable when in user or kernel mode (irrespective of the KW bit value). Note: If MPU_EN[15]==1, user mode refers to the secure user mode only and kernel mode refers to the secure kernel mode only. If MPU_EN[15]==0, the user mode can be a normal user mode or the secure user mode and kernel mode can be secure kernel mode or the normal kernel mode.)
UR	[5]	General Read Permission 0: Default protection region is read-protected in user mode. 1: Default protection region is readable in user or kernel mode (irrespective of the KR bit value). Note that the UR field controls only read access; instruction-fetch access is controlled by the UE field. Note: If MPU_EN[15]==1, user mode refers to the secure user mode only and kernel mode refers to the secure kernel mode only. If MPU_EN[15]==0, the user mode can be a normal user mode or the secure user mode and kernel mode can be secure kernel mode or the normal kernel mode.)
KE	[6]	Kernel-only Execute Permission 0: Instruction fetch is not permitted in default protection region. 1: Instruction fetch is permitted to the default protection region when in kernel mode only.
KW	[7]	Kernel-only Write Permission 0: Default protection region is write-protected. 1: Default protection region is writable when in kernel mode only. Note: If MPU_EN[15]==1, kernel mode refers to the secure kernel mode only. If MPU_EN[15]==0, kernel mode can be secure kernel mode or the normal kernel mode.)
KR	[8]	Kernel-only Read Permission 0: Default protection region is read-protected. 1: Default protection region is readable when in kernel mode only. Note that the KR field controls only read access; instruction-fetch access is controlled by the KE field. Note: If MPU_EN[15]==1, kernel mode refers to the secure kernel mode only. If MPU_EN[15]==0, kernel mode can be secure kernel mode or the normal kernel mode.)

Table 18-6 MPU Enable Register

Field	Bit	Description
S	[15]	Specifies if the memory region is normal or secure 0: normal memory region 1: secure memory region This field is only valid when MPU_BUILD[S] bit is 0x1.
SID[7:0]	[23:16]	Default region SID field; the default value is 0x1. This field is only valid when MPU_BUILD[S] bit is 0x1. If -mpu_sid_option==false, these bits are read as zero and ignored on writes.
EN	[30]	MPU enable bit - always enabled when the -sec_modes_option==true A value of 0 means disabled; 1 means enabled.

18.1.8 MPU Exception Cause Register, MPU_ECR

Address: 0x420

Access: R

The MPU_ECR register records the memory region that caused the most recent exception and indicates the type of transaction causing the exception.

This register contains 0x00060000 on reset.

The Protection Violation exception, EV_ProtV, is used to signal an MPU violation. This exception is used to also signify a stack out-of-bounds exception or a code protection violation. The EC_CODE field is updated by all of these exceptions, and uses the parameter field (bits 7:0) to indicate the origin of the exception (MPU, Code Protection, Stack out-of-bounds, and so on). The MR field indicates the actual region selected by a faulting address. The special value 0xff for this MR field indicates that no region match was detected.

Figure 18-24 MPU_ECR Register

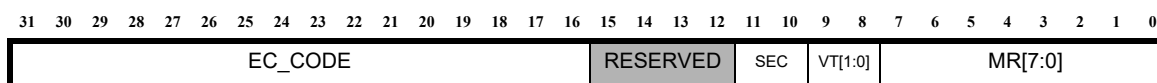


Table 18-7 MPU Exception Cause Register

Field	Bit	Description
MR	[7:0]	MPU Region Indicates which protection region was violated to cause the interrupt according to the following values: ■ 0x00 MPU region 0 ■ 0x01 MPU region 1 ■ 0x02 MPU region 2 ■ 0x03 MPU region 3 ■ 0x04 MPU region 4 ■ 0x05 MPU region 5 ■ 0x06 MPU region 6 ■ 0x07 MPU region 7 ■ 0x08 MPU region 8 ■ 0x09 MPU region 9 ■ 0x0a MPU region 10 ■ 0x0b MPU region 11 ■ 0x0c MPU region 12 ■ 0x0d MPU region 13 ■ 0x0e MPU region 14 ■ 0x0f MPU region 15 ■ 0xff Access missed all regions
VT	[9:8]	Violation Type Indicates the type of access that caused the violation according to the following values: ■ 0x0: Execution Violation ■ 0x1: Data Read Violation ■ 0x2: Data Write Violation ■ 0x3: Data Read-modify-write Violation (such as EX instruction)
SEC	[11:10]	Indicates if a secure violation has occurred. This field is only valid when MPU_BUILD[S] bit is 0x1. ■ 0x0: No secure violation has occurred. ■ 0x1: Secure violation has occurred; secure resources are being accessed in the normal mode. ■ 0x2: An SID mismatch has occurred. ■ 0x3: An SID and secure mode mismatch have occurred.
EC_CODE[15:0]	[31:16]	Vector number of the EV_ProtV exception: set to 0x0006

18.1.9 MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB15

Address: MPU_RDB0: 0x422, MPU_RDB1: 0x424
MPU_RDB15: 0x440

Access: RW

Reset 0x00000000

The MPU region descriptor base registers contain the base address for each region and enables the particular region.

The MPU version 0x02 supports a minimum protection region size of 2 Kbytes.

The MPU version 0x03 supports a minimum protection region size of 32 bytes.

Any MPU region that is programmed with an undefined region size assumes the minimum region size. If the base address is not a multiple of the region size, the effective base address is the highest address that is less than the given address and which is also a multiple of the region size. In other words, the base address is the address of the region containing the programmed address.

Figure 18-25 MPU Region Descriptor Base Registers

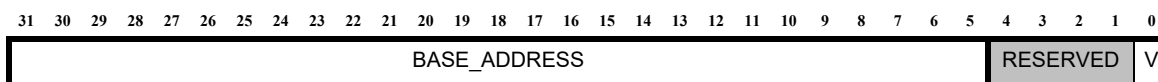


Table 18-8 MPU Region Descriptor Base Registers Field Description

Field	Bit	Description
V	[0]	Region Descriptor Validity Flag ■ 0: This region descriptor is unused. ■ 1: This region descriptor is valid.
BASE_ADDRESS	[31:5]	Base Address Logical base address of the region; must be aligned to a multiple of the region size The MPU version 0x02 supports a minimum protection region size of 2 Kbytes. The base address will occupy bits [31:11]. The MPU version 0x03 and later support a minimum protection region size of 32 bytes.

18.1.10 MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP15

Address: MPU_RDP0: 0x423, MPU_RDP1: 0x425 ...
MPU_RDP15: 0x441

Access: RW

Reset 0x00000000

The MPU region descriptor permissions registers set the size and set the permissions for each region. This register contains 0x00000000 on reset.

Figure 18-26 MPU Region Descriptor Permissions Registers

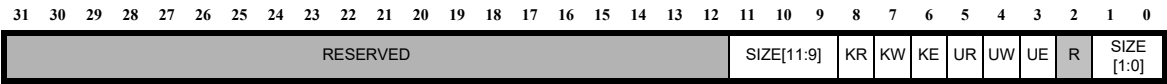


Table 18-9 MPU Region Descriptor Permission Registers Field Description

Field	Bit	Description																														
SIZE	[11:9] and [1:0]	Size; The size of the region is a 5-bit field, the three MSB bits are represented in [11:9] and the two LSB bits are represented in [1:0]. Together these fields specify the size of the region in bytes: Note: For MPU version 0x03, minimum region size is 32 bytes. For MPU version 0x02, minimum region size is 2 Kbytes. <table><tr><td>00000-00011</td><td>Reserved</td></tr><tr><td>0x4 32</td><td>0x5 64</td></tr><tr><td>0x6 128</td><td>0x7 256</td></tr><tr><td>0x8 512</td><td>0x9 1k</td></tr><tr><td>0xA 2K</td><td>0xB 4K</td></tr><tr><td>0xC 8K</td><td>0xD 16K</td></tr><tr><td>0xE 32K</td><td>0xF 64K</td></tr><tr><td>0x10 128K</td><td>0x11 256K</td></tr><tr><td>0x12 512K</td><td>0x13 1M</td></tr><tr><td>0x14 2M</td><td>0x15 4M</td></tr><tr><td>0x16 8M</td><td>0x17 16M</td></tr><tr><td>0x18 32M</td><td>0x19 64M</td></tr><tr><td>0x1A 128M</td><td>0x1B 256M</td></tr><tr><td>0x1C 512M</td><td>0x1D 1G</td></tr><tr><td>0x1E 2G</td><td>0x1F 4G</td></tr></table> Note: The MPU region size has the following constraints: ■ MPU region size >= minimum cache line size of the L1 cache ■ MPU region size >= minimum cache line size of the L2 cache	00000-00011	Reserved	0x4 32	0x5 64	0x6 128	0x7 256	0x8 512	0x9 1k	0xA 2K	0xB 4K	0xC 8K	0xD 16K	0xE 32K	0xF 64K	0x10 128K	0x11 256K	0x12 512K	0x13 1M	0x14 2M	0x15 4M	0x16 8M	0x17 16M	0x18 32M	0x19 64M	0x1A 128M	0x1B 256M	0x1C 512M	0x1D 1G	0x1E 2G	0x1F 4G
00000-00011	Reserved																															
0x4 32	0x5 64																															
0x6 128	0x7 256																															
0x8 512	0x9 1k																															
0xA 2K	0xB 4K																															
0xC 8K	0xD 16K																															
0xE 32K	0xF 64K																															
0x10 128K	0x11 256K																															
0x12 512K	0x13 1M																															
0x14 2M	0x15 4M																															
0x16 8M	0x17 16M																															
0x18 32M	0x19 64M																															
0x1A 128M	0x1B 256M																															
0x1C 512M	0x1D 1G																															
0x1E 2G	0x1F 4G																															
UE	[3]	General Execute Permission ■ 0x0: Instruction fetch is not permitted. ■ 0x1: Instruction fetch is permitted when in user or kernel mode.																														
UW	[4]	General Write Permission ■ 0x0: Region is write-protected. ■ 0x1: Region is writable when in user or kernel mode.																														
UR	[5]	General Read Permission ■ 0x0: Region is read-protected. ■ 0x1: Region is readable when in user or kernel mode. Note that the UR field controls only read access; instruction-fetch access is controlled by the UE field.																														
KE	[6]	Kernel-only Execute Permission ■ 0x0: Instruction fetch is not permitted. ■ 0x1: Instruction fetch is permitted when in kernel mode only.																														

Table 18-9 MPU Region Descriptor Permission Registers Field Description

Field	Bit	Description
KW	[7]	Kernel-only Write Permission 0x0: Region is write-protected. 0x1: Region is writable when in kernel mode only.
KR	[8]	Kernel-only Read Permission 0x0: Region is read-protected. 0x1: Region is readable when in kernel mode only. Note that the KR field controls only read access; instruction-fetch access is controlled by the KE field.

**Caution**

A region descriptor must not be enabled unless all fields have been set to well-defined values. For example, the region size must be set to a non-reserved value, and all reserved bits must be zero. You must disable the MPU before any region descriptors are updated by setting the MPU_EN register to 0x00000000. After all of the region descriptors have been set to well-defined values, the MPU can be re-enabled by setting MPU_EN to 0x40000000.

Any MPU region that is programmed with an undefined region size assumes the minimum region size. If the base address is not a multiple of the region size, the effective base address is the highest address that is less than the given address and is also a multiple of the region size. In other words, the base address is the address of the region containing the programmed address.

Table 18-10 Build Configuration Registers

Number	Name	Access	Description
0x6D	Memory Protection Build Configuration Register, MPU_BUILD	R	Build configuration for: memory protection unit

18.1.11 Memory Protection Build Configuration Register, MPU_BUILD

Address: 0x6D

Access: R

The MPU_BUILD register can be used to detect the presence of an MPU, and to determine the number of supported regions.

Figure 18-27 MPU_BUILD Register

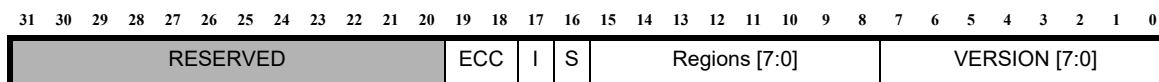


Table 18-11 MPU_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version Indicates the version of the MPU: 0x0 : When no MPU is present 0x2 : MPU with 2 Kbytes minimum region size 0x3 : MPU with 32 bytes minimum region size 0x04 : Supports defining regions as secure and normal, added SID protection, indirect programming interface; when <code>-sec_modes_option==true</code> 0x8: Support independence of kernel and user mode in secure MPU; support for ECC protection on MPU auxiliary registers (bits [19:18] determines the configuration) All other values are reserved.
Regions	[15:8]	Number of Regions Indicates the number of regions supported: 0x00 when no MPU is present 0x01 for 1 region 0x02 for 2 regions 0x04 for 4 regions 0x08 for 8 regions 0x10 for 16 regions All other values are reserved.
S	[16]	MPU supports defining regions as secure or normal. This bit exists only when MPU version is 0x4 or 0x8.

Table 18-11 MPU_BUILD Field Description

Field	Bit	Description
I	[17]	MPU supports SID mechanism. This bit exists only when MPU_BUILD[S] bit is 0x1. 0x0: SID is not supported; -mpu_sid_option==false 0x1: SID is supported; -mpu_sid_option==true
ECC	[19:18]	Error protection on MPU auxiliary registers. This field is valid from MPU version 0x8. 0x0: no protection 0x1: SEDDED (single-bit and double-bit error detection) 0x2: SECDED (single-bit error detection and correction, double-bit error detection)

18.1.12 MPU Exceptions

See [Table 6-6](#) on page 268.

19

Protection Schemes

ARCV2 ISA-based processors provide several memory-protection options, which are described in this chapter. See the Databook for your specific processor to determine which options are available. These protection options include:

- Stack protection: checks overflow and underflow of reserved stack space
- Code protection: hardware means to block read and write to code space

The different protection schemes may be combined to achieve several levels of protection against malicious or misbehaving code in critical applications.

19.1 Stack Checking

Stack checking is a mechanism for checking stack accesses and raising an exception when a stack overflow or underflow is detected, provided stack exceptions are enabled. The logic triggers an `EV_ProtV` exception when a violation occurs. Topics covered include configuring this option for your core, enabling the option at runtime, and programming the stack range. If stack exceptions are enabled, the memory region in which the stack is located can be explicitly programmed using the following optional auxiliary registers; two for kernel and two for user mode:

- `USTACK_TOP`, 0x260
- `USTACK_BASE`, 0x261
- `KSTACK_TOP`, 0x264
- `KSTACK_BASE`, 0x265

The SC bit is cleared on exception entry and restored on exception exit when the `STATUS32` register is restored. The SC bit is unchanged on interrupt entry.

Stack checking is a mechanism for checking stack accesses and raising an exception when a stack overflow or underflow is detected, provided stack exceptions are enabled. The logic triggers an `EV_ProtV` exception when a violation occurs. If stack exceptions are enabled, the memory region in which the stack is located can be explicitly programmed using the following optional auxiliary registers; two for kernel and two for user mode:

- `USTACK_TOP`, 0x260
- `USTACK_BASE`, 0x261

- KSTACK_TOP, 0x264
- KSTACK_BASE, 0x265

When stack checking exceptions are raised in the normal mode, the SC bit is cleared on exception entry and restored on exception exit when the STATUS32 register is restored. The SC bit is unchanged on interrupt entry.

If the processor includes the secure mode (`-sec_modes_option==true`), stack checking is configured in the secure mode, and the memory region in which the stack is located can be explicitly programmed using the following optional auxiliary registers; two for secure kernel and two for secure user mode:

- S_USTACK_TOP, 0x262
- S_USTACK_BASE, 0x263
- S_KSTACK_TOP, 0x267
- S_KSTACK_BASE, 0x266

When stack checking exceptions are raised in the secure mode, the SSC bit is cleared on exception entry and restored on exception exit when the SEC_STAT register is restored. The SSC bit is unchanged on interrupt entry.

19.1.1 Build Configuration Register

[Stack Region Configuration Register, STACK_REGION_BUILD](#) (0xC5) identifies the presence and version of stack checking in the build.

19.1.2 Enabling Stack Checking

The kernel can enable stack checking by setting the SC bit (bit 14) in the [Status Register, STATUS32](#). In user mode, this bit is not writable and returns zero on read. If stack checking is not configured for the core, this bit is not writable and returns zero on read. In user mode, the STATUS32.SC bit is read as zero and ignored on writes. When stack checking is included in the configuration, the following auxiliary registers are included in the processor:

- KSTACK_BASE, KSTACK_TOP, USTACK_BASE, and USTACK_TOP
- STACK_REGION_BUILD BCR

Enable stack checking in the normal kernel or user mode by setting the SC bit (bit 14) in the [Status Register, STATUS32](#). In user mode, this bit is not writable and returns zero on read. If stack checking is not configured for the core, this bit is not writable and returns zero on read. When stack checking is included in the configuration, the following auxiliary registers are included in the processor:

- KSTACK_BASE, KSTACK_TOP, USTACK_BASE, and USTACK_TOP
- STACK_REGION_BUILD BCR

Enable stack checking in the secure kernel or user mode by setting SEC_STAT.SSC==1. In user modes, the SEC_STAT.SSC bit is read as zero and ignored on writes. When `-sec_modes_option==true`, the following auxiliary registers are included in the processor:

- S_KSTACK_BASE, S_KSTACK_TOP, S_USTACK_BASE, and S_USTACK_TOP
- STACK_REGION_BUILD BCR

19.1.3 Specifying Stack Regions

Specify the memory regions reserved for the stack by loading their addresses into the following auxiliary registers. Separate register pairs define the kernel-mode and user-mode stack regions.

**Note**

The stack grows downward from higher memory address to a lower memory address.

- `USTACK_TOP` (0x260) is the user-mode lower address limit.
- `USTACK_BASE` (0x261) is the user-mode upper address limit.
- `KSTACK_TOP` (0x264) is the kernel-mode lower address limit.
- `KSTACK_BASE` (0x265) is the kernel-mode upper address limit.
- `S_USTACK_TOP` (0x262) is the secure user-mode lower address limit.
- `S_USTACK_BASE` (0x263) is the secure user-mode upper address limit.
- `S_KSTACK_TOP` (0x266) is the secure kernel-mode lower address limit.
- `S_KSTACK_BASE` (0x267) is the secure kernel-mode upper address limit.

All stack operations are performed on 32-bit words, so write the lower two bits of the address as zero. The number of valid bits is limited to the configured address size of the processor.

19.1.4 Stack-Checking Operation

The reserved stack region can be accessed by explicit instructions (such as LD or ST) or by implicit operations (such as an interrupt sequence).

The stack-checking logic differentiates between explicit instructions, which use %SP as the base register and other instructions, which use other addressing modes to access the allocated stack region. This distinction achieves functional safety while allowing common compiler operations to take place.

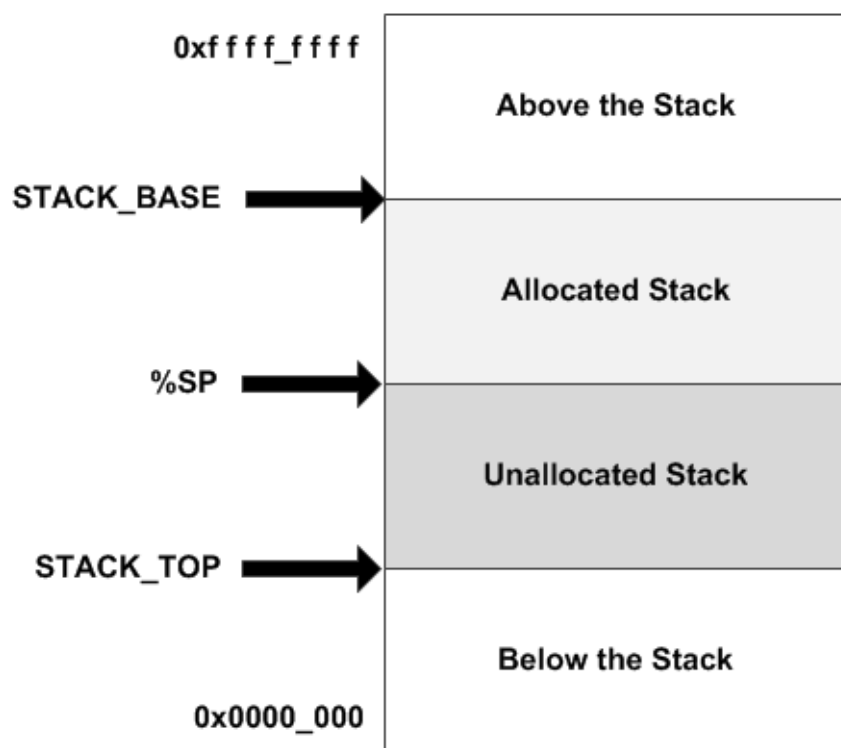
19.1.4.1 Stack Regions

Figure 19-1 shows the stack regions, as defined by the range registers and the architectural stack-pointer register (%SP). The allocated and unallocated regions make up the reserved stack space. The stack grows downward.

- %SP is the architectural stack-pointer register, which usually points to the top of the stack. It is always r28 in ARCV2-based processors.
- The allocated stack region is defined as all addresses from (and including) %SP up to (but excluding) STACK_BASE.
- The unallocated stack region is defined as all addresses from (and including) STACK_TOP up to (but excluding) %SP.
- STACK_BASE points to the first 32-bit word above the reserved stack space, and therefore all addresses greater than or equal to STACK_BASE are above the stack.
- STACK_TOP points to the last allocatable word in the stack region, and therefore all addresses less than STACK_TOP are below the stack.

**Note**

The stack grows downward from higher memory address to a lower memory address.

Figure 19-1 Stack regions

19.1.4.2 Stack Access Rules

- Non-%SP based memory references may not reference the unallocated stack region.
- %SP-based memory references that do not modify %SP must be within the allocated stack region.
- %SP-based memory references that modify %SP must be within the reserved (allocated and unallocated) stack space.

Violation of one of these rules when stack checking is enabled results in an EV_ProtV exception.

Figure 19-2 and Figure 19-2 illustrate the conditions checked in three possible scenarios.

Figure 19-2 Stack Checking Scenarios

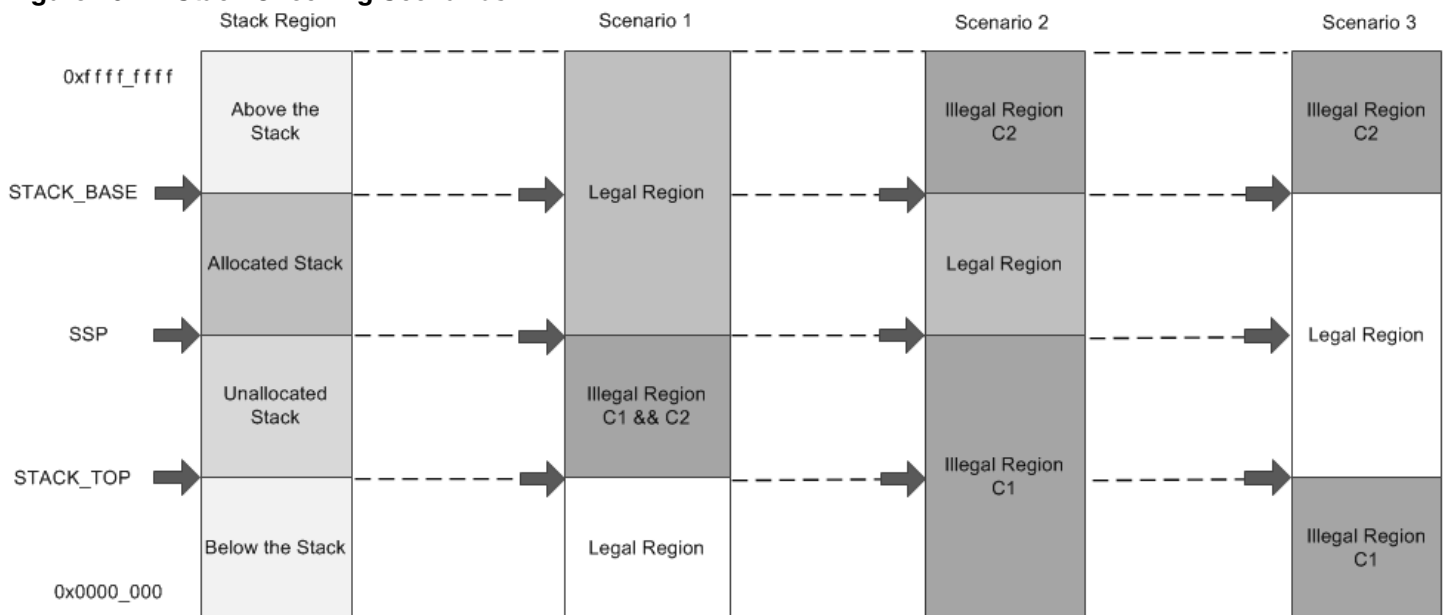


Table 19-1 Stack Checking Specification

	Scenario 1 Non-%SP-Based Address. Example: LD r0, [r1];	Scenario 2 %SP-Based, No Auto-Update of %SP Example: LD r0, [r28];	Scenario 3 %SP-Based, with Auto-Update of %SP Example: LD.a r0, [r28];
Conditions	Example: LD r0, [r1];	Example: LD r0, [r28];	Example: LD.a r0, [r28];
Condition C1	address < SSP	address < SSP	address < STACK_TOP
Condition C2	address ≥ STACK_TOP	address ≥ STACK_BASE	address ≥ STACK_BASE
Condition C3		address < STACK_TOP	
Error Condition	C1 && C2	C1 C2 C3	C1 C2

19.1.5 Exception Information

An exception caused by a stack checking violation has the following attributes:

- Exception: EV_ProtV
 - Vector Number: 0x06
 - Vector offset: 0x18
 - Cause code determined by failed reference type
 - Parameter 0x02

On entry to the EV_ProtV exception, the ECR register is set according to the [Table 6-6](#) on page 268

19.1.6 Stack Protection Out-of-Bound Limitations

Any multi-cycle instruction with SP as target has SP update pending till instruction is complete. For example DIV, REM instructions with SP as destination. Any use of SP when SP update is pending is not checked for Stack Checking violation.

19.1.7 Stack Region Auxiliary Register Set

Table 19-2 Stack Region Checking Auxiliary Registers

Auxiliary Register	Name	Description
0x260	USTACK_TOP	User stack top address register
0x261	USTACK_BASE	User stack base address register
0x262	S_USTACK_TOP (see S_STACK_TOP)	Secure mode user stack top address register
0x263	S_USTACK_BASE (see S_STACK_BASE)	Secure mode user stack base address register
0x264	KSTACK_TOP	Kernel stack top address register
0x265	KSTACK_BASE	Kernel stack base address register
0x266	S_KSTACK_TOP (see S_STACK_TOP)	Secure mode kernel stack top address register
0x267	S_KSTACK_BASE (see S_STACK_BASE)	Secure mode kernel stack base address register

19.1.7.1 User Stack Region Top Address, USTACK_TOP

USTACK_TOP Address:	0x260
Access	RW
Reset:	0x0

You can use the following auxiliary registers to set the stack region in the normal user mode. This auxiliary register is present in the processor if you have configured the processor to include stack checking.

Figure 19-3 USTACK_TOP when ADDR_SIZE == 32

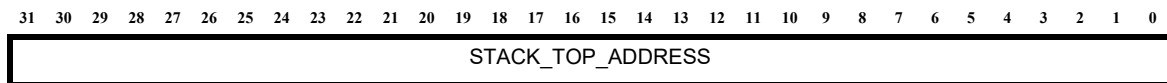
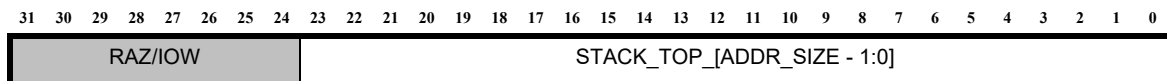


Figure 19-4 USTACK_TOP when ADDR_SIZE < 32



You can use the stack region top address register, USTACK_TOP to set the top address of the valid stack region in memory.

All load and store addresses are checked against the region between USTACK_BASE and USTACK_TOP. Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store is between the SP and USTACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the USTACK_BASE to USTACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

19.1.7.2 User Stack Region Base Address, USTACK_BASE

USTACK_BASE Address: 0x261

Access RW

Reset: 0x0

Figure 19-5 USTACK_BASE when ADDR_SIZE == 32

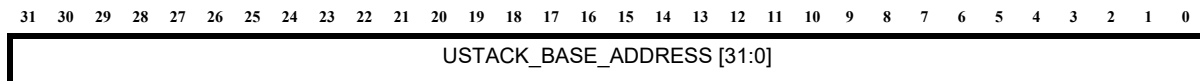
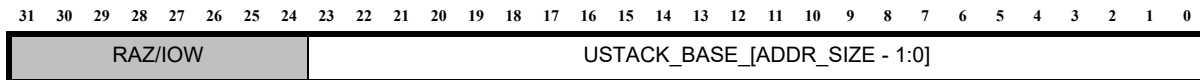


Figure 19-6 USTACK_BASE when ADDR_SIZE < 32



The stack region base address register, USTACK_BASE, is a read or write register, and is used to set the base address of the valid stack region in memory.

You can use the following auxiliary registers to set the stack region in the normal user mode. This auxiliary register is present in the processor if you have configured the processor to include stack checking.



Note

The stack grows downward from higher memory address to a lower memory address.

All load and store addresses are checked against the region between USTACK_BASE and USTACK_TOP. Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store accesses the region between the SP and USTACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the USTACK_BASE to USTACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

19.1.7.3 Kernel Stack Region Top Address, KSTACK_TOP

KSTACK_TOP Address: 0x264

Access RW

Reset: 0x0

Figure 19-7 KSTACK_TOP when ADDR_SIZE == 32

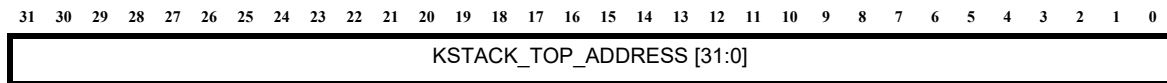
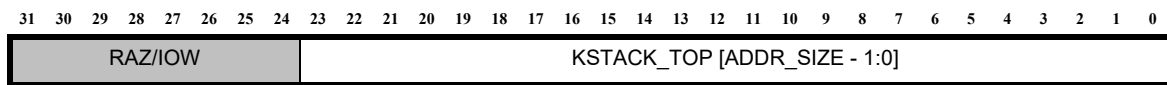


Figure 19-8 KSTACK_TOP, when ADDR_SIZE < 32



You can use the stack region top address register, KSTACK_TOP to set the top address of the valid stack region in memory in kernel mode.

You can use the following auxiliary registers to set the stack region in the normal kernel mode. This auxiliary registers is present in the processor if you have configured the processor to include stack checking.

All load and store addresses are checked against the region between KSTACK_BASE and KSTACK_TOP. Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store is between the SP and KSTACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the KSTACK_BASE to KSTACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

19.1.7.4 **Kernel Stack Region Base Address, KSTACK_BASE**

KSTACK_BASE Address:	0x265
Access	RW
Reset:	0x0

Figure 19-9 KSTACK_BASE when ADDR_SIZE == 32

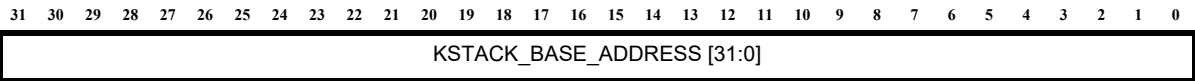
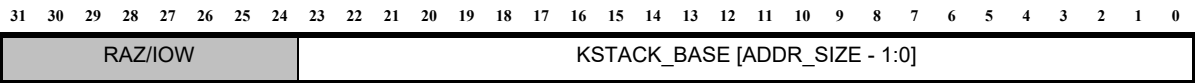


Figure 19-10 KSTACK_BASE when ADDR_SIZE < 32



The stack region base address register, KSTACK_BASE, is a read or write register, and is used to set the base address of the valid kernel stack region in memory.

You can use the following auxiliary registers to set the stack region in the normal kernel mode. This auxiliary register is present in the processor if you have configured the processor to include stack checking.

 **Note**

The stack grows downward from higher memory address to a lower memory address.

All load and store addresses are checked against the region between KSTACK_BASE and KSTACK_TOP. Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store accesses the region between the SP and KSTACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the KSTACK_BASE to KSTACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

19.1.7.5 Secure Stack Region Top Address, S_STACK_TOP

S_USTACK_TOP Address: 0x262

S_KSTACK_TOP Address: 0x266

Access RW in the secure mode only

You can use the stack region top address register, S_STACK_TOP to set the top address of the valid stack region in the memory. These registers are used for stack checking only when the core is in the secure user or kernel mode.

Figure 19-11 S_USTACK_TOP when ADDR_SIZE == 32

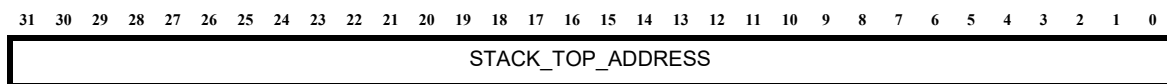


Figure 19-12 S_USTACK_TOP when ADDR_SIZE < 32

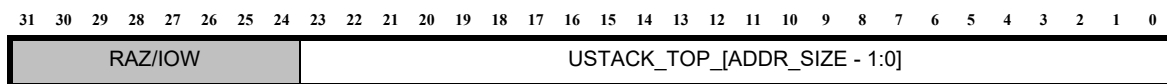


Figure 19-13 S_KSTACK_TOP when ADDR_SIZE == 32

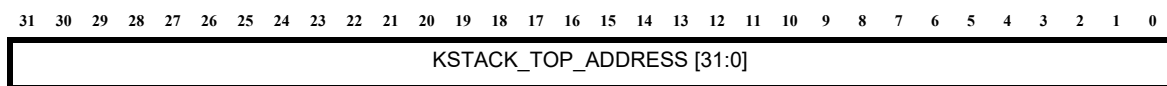
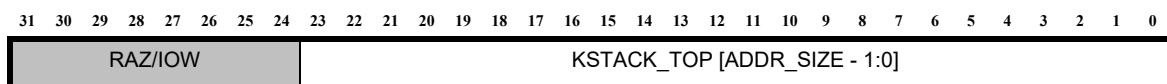


Figure 19-14 S_KSTACK_TOP, when ADDR_SIZE < 32



All load and store addresses are checked against the region between STACK_BASE and STACK_TOP. Set the SEC_STAT.SSC==1 to enable stack checking in the secure mode.

If any load or store accesses the region between the SP and STACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the STACK_BASE to STACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

19.1.7.6 Secure Stack Region Base Address, S_STACK_BASE

S_USTACK_BASE Address: 0x263

S_KSTACK_BASE Address: 0x267

Access RW in the secure mode only

The secure stack region base address register, a read or write register in the secure kernel mode, is used to set the base address of the valid stack region in the memory when the core is operating in a secure user or kernel mode. When the core is in the secure user or kernel mode, these registers are used for stack checking. These auxiliary registers are present in the processor if you have configured `-sec_modes_option==true`.



Note

The stack grows downward from higher memory address to a lower memory address.

Figure 19-15 S_USTACK_BASE when ADDR_SIZE == 32

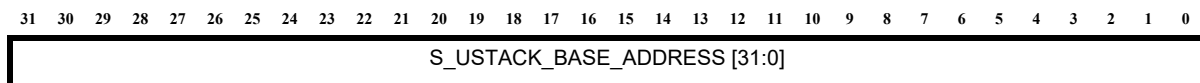


Figure 19-16 S_USTACK_BASE when ADDR_SIZE < 32

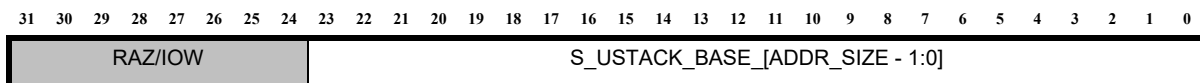


Figure 19-17 S_KSTACK_BASE when ADDR_SIZE == 32

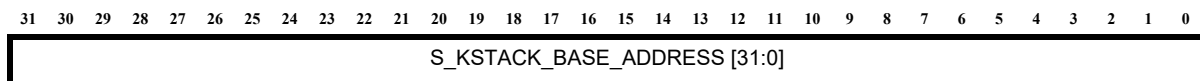
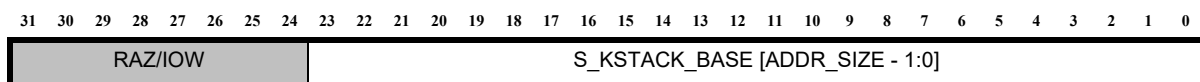


Figure 19-18 S_KSTACK_BASE when ADDR_SIZE < 32



All load and store addresses are checked against the region between STACK_BASE and STACK_TOP. Set the SEC_STAT.SSC==1 to enable stack checking in the secure mode.

If any load or store accesses the region between the SP and STACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the STACK_BASE to STACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

19.1.7.7 Stack Region Configuration Register, STACK_REGION_BUILD

Address: 0xC5

Access: R

The stack region configuration register, STACK_REGION_BUILD, indicates the presence of the stack region registers, [User Stack Region Base Address, USTACK_BASE](#) and [User Stack Region Top Address, USTACK_TOP](#).

Figure 19-19 STACK_REGION_BUILD Register

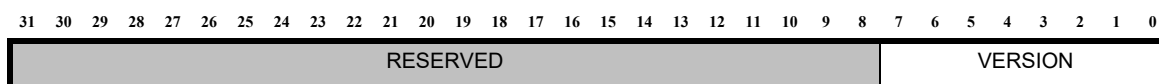


Table 19-3 STACK_REGION_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of stack region checking ■ 0x1: ARC 700 style stack checking ■ 0x2: ARCV2 style stack checking
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

19.1.8 Stack Checking Exceptions

See [Table 6-6](#) on page [268](#).

19.2 Coexisting Protection Schemes

Stack Checking can coexist with Code Protection, MPU.

- Each protection scheme applies its own specific protection rules to all relevant memory references.
- A protection violation exception is raised if any coexistent protection scheme detects a protection violation.
- If multiple protection schemes raise exceptions simultaneously, they all share the same vector and cause code. However, each protection scheme sets its own distinct bit in the 8-bit parameter field, as follows:
 - Code Protection violations set bit 0
 - Stack Checking violations set bit 1
 - MPU permission violations set bit 2

Therefore, multiple protection violations can be reported by raising a single exception. See the [Table 6-6](#) for more information about the exception information for each of the protection schemes.

19.3 Code Protection

This chapter describes the optional code protection mechanism in the ARCV2 processor.

If code protection is configured in the processor, the `CPROT_BUILD` register is added to the processor. The software can read this register to determine the code protection unit configuration.

The code protection option is intended to prevent piracy and hacking of code memory by blocking unauthorized loads and stores without impacting processor performance.

19.3.1 Code Protection Scheme

The ARCV2 ISA architecture divides the memory into 16 regions, which can be protected individually.

This feature adds a 16-bit input to the processor core, one bit per region. When the protect bit is set, the processor disables any load or store to the corresponding region. An attempt to access a protected region raises an `EV_ProtV` exception. The protect input bits must remain stable when the processor is running.



Attention

This feature offers only basic protection, and is not a substitute for advanced protection methods. Code residing in external memory is potentially exposed during instruction fetch.

19.3.2 Code Protection Behavior

This section describes how the code protection scheme behaves in relation to specific scenarios.

19.3.2.1 Processor Reset

Protection is valid immediately after reset.

19.3.2.2 User and Kernel Mode

The code protection scheme applies to both user and kernel modes.

19.3.2.3 Stack Location

For proper program execution, ensure that the stack is not placed in a code-protected region.

If the stack enters a protected region during automatic context save or restore on interrupt entry or exit, the interrupt is not taken, and an `EV_ProtV` exception is raised. If the processor is already in exception mode (`STATUS32.AE == 1`) then a `MachineCheck` exception is raised.

19.3.2.4 DMI Access

DMI access violations do not raise code-protection exceptions. Write commands from DMI are ignored, and read transactions return zero data.

19.3.2.5 Debug

Code-protection violations do not raise exceptions during debug access. The processor merely ignores load and store instructions in code-protected regions. This scheme allows single-stepping in a code-protected region.

19.3.2.6 DFT Scan

Code protection is not available when the core is accessed through the DFT (Design-for-Test) scan chain.

19.3.2.7 Code Protection Exception

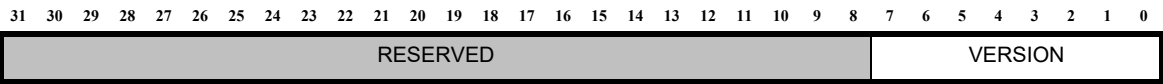
The `EV_ProtV` exception is taken, the same as other data-access protection violations.

For details on the ECR register, see the [Exception Cause Register, ECR](#).

19.3.3 Code Protection Register, CPROT_BUILD

Address: 0xC9
Access: R

Figure 19-20 CPROT_BUILD Register



The code protection register specifies if code protection is enabled or disabled in the processor configuration. This register is present only if the code protection unit is part of the processor build.

Table 19-4 CPROT_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Code protection feature 0x0: code protection feature is disabled 0x1: code protection feature is enabled

20

ICCM

20.1 ICCM Auxiliary Registers

This section describes the ARCV2 Instruction Closely Coupled Memories (ICCM0 and ICCM1). ICCMs are designed to hold performance-critical code.

The ARCV2 architecture includes the following auxiliary registers and build configuration registers when the instruction close coupled memory is enabled. In builds which do not have any means to fetch instructions other than through the multiple ICCMs, the starting address of the ICCM0 is fixed at address 0.

Table 20-1 ICCM Registers

Address	Auxiliary Register Name	Description
0x208	AUX_ICCM	Base address of multiple ICCMs
0x20F	CCM_PART_SIZE	Specifies the size of the ICCM0 partition
0x78	ICCM_BUILD	Build configuration register for ICCM

20.1.1 ICCM base address, AUX_ICCM

Address: 0x208

- Access:
- When SecureShield 2+2 mode is not configured: RW
 - When SecureShield 2+2 mode is configured: RW in the secure mode only

Figure 20-1 AUX_ICCM, base address for ICCM (ADDR_SIZE == 32)



Figure 20-2 AUX_ICCM, base address for ICCM (ADDR_SIZE == 16)

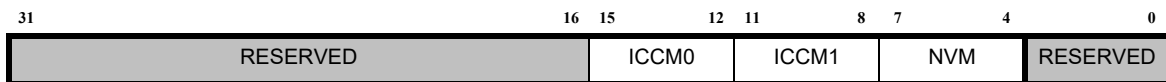
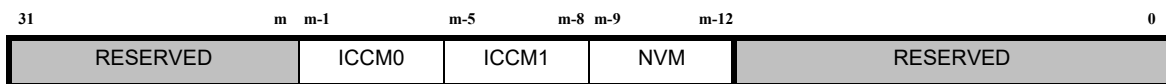


Figure 20-3 AUX_ICCM, base address for ICCM (ADDR_SIZE == m)



If an ICCM0, ICCM1, or NV_ICCM is not configured in a build, the corresponding base address field of the AUX_ICCM register is reserved (RAZ/IOW). The ICCMs can occupy memory regions zero through 7. If ICCM0 is configured, the AUX_ICCM.ICCM0 field identifies the region that contains the ICCM0. If ICCM1 is configured, the AUX_ICCM.ICCM1 field identifies the region that contains the ICCM1. If NV_ICCM is configured, the AUX_ICCM.NVM field identifies the region that contains the NV_ICCM. The most-significant relevant bit of AUX_ICCM is determined by the configured size of the address bus ADDR_SIZE, such that the 4-bit Region number for the ICCM0 is always located at positions [ADDR_SIZE-1:ADDR_SIZE-4] and ICCM1 is always located at positions [ADDR_SIZE-5:ADDR_SIZE-8], and NV_ICCM is always located at positions

[ADDR_SIZE-9:ADDR_SIZE-12]. For proper utilization of the memory space, ICCMs must be spaced in such a way that the ICCM addresses do not overlap. If they overlap, ICCM0 has highest priority to get access to the regions where they overlap, ICCM1 has higher priority to get access to the regions where they overlap than NV_ICCM.

In systems with both ICCM and external memory, these memories may overlap. When fetching instructions from an address at which such an overlap occurs, the ICCM is always accessed in preference to the external memory. If instruction cache, IFQ, ICCM0, and ICCM1 do not exist in the processor design, NV_ICCM is always be selected for fetch access no matter the memory region.

Repositioning the ICCM does not affect the contents of external memory. Similarly, external DMA accesses to addresses, overlapping external memory and ICCM, do not affect the contents of ICCM.

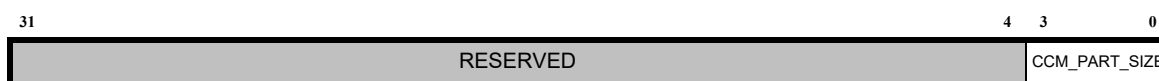
20.1.2 ICCM Partition Size Register, CCM_PART_SIZE

Address: 0x20F

Access: R

Default 0x0

Figure 20-4 CCM_PART_SIZE Register



This register specifies the size of the first partition in the ICCM. This register exists only when `ICCM_BUILD >= 0x7`.

Table 20-2 CCM_PART_SIZE Field Descriptions

Field	Bit	Reset	Description
CCM_PART_SIZE	[3:0]	0	Size of ICCM0 partition 0x0: no partition 0x2: 1 KB 0x3: 2 KB 0x4: 4 KB 0x5: 8 KB 0x6: 16 KB Other values are reserved
RESERVED	[31:4]	0	These bits read as zero.

Following are the constraints on the ICCM0-R memory:

- `iccm0_partition_size` must be less than `iccm0_size`.
- `iccm0_prot` must be set to `None`, if `iccm0_partition_size` is greater than 0.
- `iccm0_partition_size` must be 0 if secure features are configured in the processor.
- `iccm0_partition_size` must be 0, if safety manager is configured in the processor.
- `iccm0_size+iccm0_partition_size` must be less than region size.

20.1.3 ICCM Configuration Register, ICCM_BUILD

Address: 0x78

Access: R

The ICCM Configuration register (ICCM_BUILD) describes the size and version number of the Instruction Closely Coupled Memory.

Figure 20-5 ICCM_BUILD Register

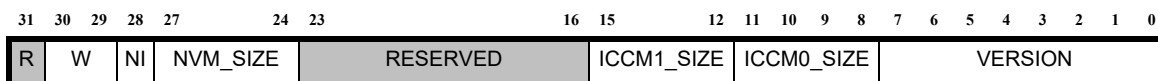


Table 20-3 ICCM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	0x4: ICCM without secure 2+2 mode configured 0x5: ICCM with secure 2+2 mode configured 0x7: Supports wide ICCM0 and ICCM0 partitioning
ICCM0_SIZE	[11:8]	Size of ICCM0 RAM 0x0: Not present 0x1: 512B 0x2: 1KB 0x3: 2KB 0x4: 4KB 0x5: 8KB 0x6: 16KB 0x7: 32KB 0x8: 64KB 0x9: 128KB 0xA: 256KB 0xB: 512KB 0xC: 1MB 0xD: 2 MB

Table 20-3 ICCM_BUILD Field Descriptions (Continued)

Field	Bit	Description
ICCM1_SIZE	[15:12]	Size of ICCM1 RAM ■ 0x0: Not present ■ 0x1: 512B ■ 0x2: 1KB ■ 0x3: 2KB ■ 0x4: 4KB ■ 0x5: 8KB ■ 0x6: 16KB ■ 0x7: 32KB ■ 0x8: 64KB ■ 0x9: 128KB ■ 0xA: 256KB ■ 0xB: 512KB ■ 0xC: 1MB ■ 0xD: 2MB
NVM_SIZE	[27:24]	■ 0x0: Not present ■ 0x1 to 0x2: Reserved ■ 0x3: 2KB ■ 0x4: 4KB ■ 0x5: 8KB ■ 0x6: 16KB ■ 0x7: 32KB ■ 0x8: 64KB ■ 0x9: 128KB ■ 0xA: 256KB
NI	[28]	Indicates how NV_ICCM is implemented ■ 0: IFQ is not implemented for NV_ICCM ■ 1: IFQ is implemented for NV_ICCM
W (ICCM0_WORD_WIDTH)	[30:29]	■ 0x0: 32bit ■ 0x1: 64 bit ■ 0x2: 128 bit ■ 0x3: Reserved

20.2 Exceptions

See [Table 6-6](#) on page 268.

21

Instruction Fetch Queue

21.1 IFQ Introduction

The Instruction-Fetch Queue (IFQ) is an optional component that prefetches instructions using short bursts on the external memory bus. This mechanism helps hide the access latency of individual instruction fetches. This component cannot be used with an instruction cache.

The IFQ is a low-cost alternative to an instruction cache. It reduces the effective access time to code residing in external memory by prefetching it ahead of the processor pipeline.

Table 21-1 IFQ Auxiliary Registers

Address	Auxiliary Register Name	Description
0x10	Invalidate Instruction Cache, IC_IVIC	Invalidate IFQ
0xFE	Instruction Fetch Queue Configuration Register, IFQUEUE_BUILD	Instruction fetch queue build configuration register

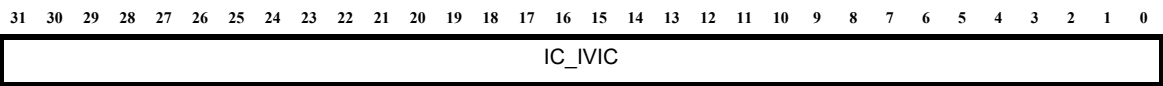
21.2 Invalidate Instruction Cache, IC_IVIC

 **Note** Only IFQ or instruction cache can be present in a build.

Address: 0x10

Access: W

Figure 21-1 IC_IVIC Register



A write to the IC_IVIC register invalidates the entire instruction cache and the instruction queue. Builds with an instruction-fetch interface to external memory or ICCM1 also utilize this register for invalidating the buffer content. For more information about this register, see [Invalidate Instruction Cache, IC_IVIC](#).

21.2.1 Instruction Fetch Queue Configuration Register, IFQUEUE_BUILD

Address: 0xFE

Access: R

The Instruction Fetch Queue Configuration register (IFQUEUE_BUILD) contains the version of the Instruction Fetch Queue.

Figure 21-2 IFQUEUE_BUILD Register

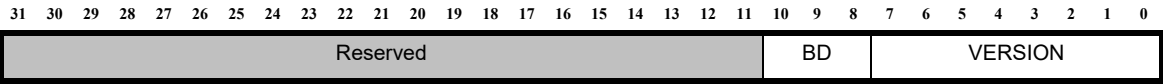


Table 21-2 IFQUEUE_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	0x02 : Current version
BD[2:0]	[10:8]	Instruction Fetch Queue entries 0x0: 1 entry 0x2: 4 entries 0x3: 8 entries All other values are reserved.

22

Instruction Cache

22.1 Instruction Cache Auxiliary Registers

The ARCV2 architecture includes additional auxiliary registers when the instruction cache is enabled. The number of additional auxiliary registers depend on the chosen `-ic_feature_level` configuration parameter. The IC_IVIC register is also included when an instruction fetch interface to external memory or ICCM1 is present in a processor. The instruction fetch path in some ARCV2-based processors may pre-fetch and cache a small number of instructions, which must be flushed when code memory is modified to ensure that stale instructions from the fetch queues are not executed. The IC_IVIC register is therefore included when either an instruction cache (I-cache) or a queuing interface to instruction memory is in the processor configuration. Writing to IC_IVIC invalidates the instruction fetch queues, instruction cache, or both.

Advanced Instruction-Cache Operation

The instruction cache contains advanced debug facilities that allow the programmer to interrogate and control the instruction cache RAM contents.

Advanced Features Summary

The instruction cache allows the programmer to obtain direct access to the internal cache RAMs through either the ARCV2 host interface bus or through LR and SR instructions. This capability provides an invaluable debug and test environment when using the instruction cache.

22.2 Auxiliary Registers

Table 22-1 Basic Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==0)

Address	Auxiliary Register Name	Description
0x10	Invalidate Instruction Cache, IC_IVIC	Invalidate instruction cache
0x11	Instruction Cache Control Register, IC_CTRL	Instruction cache control register

Table 22-2 Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==1)

Address	Auxiliary Register Name	Description
0x13	Lock Instruction Cache Line, IC_LIL	Lock instruction cache line
0x16	Invalidate Instruction-Cache Start Region, IC_IVIR	Instruction cache start region
0x17	Invalidate Instruction-Cache End Region, IC_ENDR	Instruction cache end region
0x19	Invalidate Instruction-Cache Line, IC_IVIL	Invalidate instruction cache line

Table 22-3 Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==2)

Address	Auxiliary Register Name	Description
0x1A	Instruction Cache External-Access Address, IC_RAM_ADDR	Instruction cache external access RAM address
0x1B	Instruction-Cache Tag Access, IC_TAG	Instruction cache tag access
0x1C	Instruction Cache Secure Bit Tag Register, IC_XTAG	Indicates the secure mode of a cache line in the instruction cache tag
0x1D	Instruction Cache Data Access, IC_DATA	Instruction cache data access
0x1E	Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG	Instruction Cache External Access RAM Address Physical Tag Format

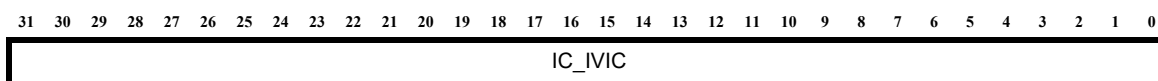
22.2.1 Invalidate Instruction Cache, IC_IVIC

Address: 0x10

Access: W

Reset: 0x0000_0000

Figure 22-1 IC_IVIC Register



When SecureShield 2+2 mode is not configured

A write to the IC_IVIC register invalidates the entire instruction cache. Builds with an instruction-fetch queue or ICCM1 also utilize this register for invalidating the buffer content.

This function is initiated by any write to the IC_IVIC auxiliary register. IC_IVIC is used to make sure the cache is coherent with memory.

When SecureShield 2+2 mode is configured

Table 22-4 IC_IVIC Description When SecureShield 2+2 mode is Configured

Operating Mode	Behavior
Core or debugger is in the normal mode	The invalidate whole cache operation is ignored.
Core or debugger is in the secure mode	A write to the IC_IVIC register invalidates the entire instruction cache. Builds with an instruction-fetch queue or ICCM1 also utilize this register for invalidating the buffer content. This function is initiated by any write to the IC_IVIC auxiliary register. IC_IVIC is used to make sure the cache is coherent with memory.



Caution

All SR accesses to the IC_IVIC register must be followed by three NOP instructions.



Caution

The IC_CTRL.SB bit is not updated when the instruction cache is successfully invalidated.

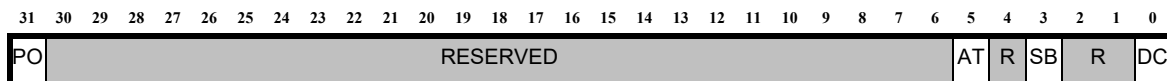
22.2.2 Instruction Cache Control Register, IC_CTRL

Address: 0x11

Access: RW

Reset: 0x0000_0000

Figure 22-2 IC_CTRL Register



The IC_CTRL register contains the control flags DC (bit 0) and AT (bit 5), and one status flag SB (bit 3).

Table 22-5 IC_CTRL Control Flags

Flag Name	Bit	Reset	Description	Access Type
DC	[0]	0	Disable Cache: Enables/Disables the cache 0x0: Enable Cache 0x1: Disable Cache	R/W
SB	[3]	0	Success Bit: Success of last cache operation SB is for compatibility with L1 instruction or data cache. SB is set (SB=1) only for the following aux operations: - Line lock: if the line cannot be locked because all ways in the sets that the address maps to are already locked, the SB flag is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when the line lock operation completes successfully. - invalidates line or region: if the system memory address is present in the cache, the cache line is invalidated and unlocked; if the line is dirty and IM=1, it is flushed to the memory subsystem. When the line is present, the SB flag is set (SB=1). If the line is not present, the SB=0.	R/W
AT	[5]	0	Address Debug Type: Used for debug purposes for when accessing cache RAMs. 0x0: Direct Cache RAM Access 0x1: Cache Controlled RAM Access	R/W
RESERVED	[31:6]	0	Reserved. Read-as-zero; ignored on write.	R

Table 22-5 IC_CTRL Control Flags

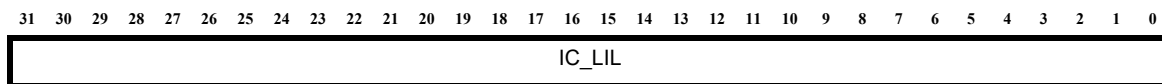
PO	[31]	0	Power Optimization enable When you enable this bit, additional RTL logic is implemented that saves power by detecting accesses to the same cache block as the previous hits, and saving signal transactions. ■ 0x0: power optimization is disabled. ■ 0x1: power optimization is enabled.	RW

For more information about the IC_CTRL register implementation, see your *ARCV2-based processor databook*.

22.2.3 Lock Instruction Cache Line, IC_LIL

Address: 0x13
 Access: W
 Reset: 0x0000_0000

Figure 22-3 IC_LIL Register



When SecureShield 2+2 mode is not configured

In non-aliased configurations, the `IC_LIL` register provides the full physical address (including the tag and index used in the cache lookup) of the line to be locked or invalidated.

In aliased instruction cache configurations, this register holds only the index bits of the line address and the `IC_PTAG` register is used to program the physical page number (PPN) of the physical page containing the line to be locked. The `IC_PTAG` register must be set-up first before triggering the operation with an SR to the `IC_LIL` register specifying the virtual index of the line to be locked. The physical byte address of the cache line is programmed in `IC_BSIZE`. Only address bits starting at bit $\log_2(\text{IC\_BSIZE})$ are significant. `IC_BSIZE` is the configured instruction cache line size.

Writing a system memory address to `IC_LIL` causes the cache to load the appropriate cache line. The line is then locked so it cannot be overwritten when a cache miss occurs. If the line cannot be locked in the cache because all of the ways in the set that the address maps to are already locked, the SB flag in `IC_CTRL` is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when an `IC_LIL` operation completed successfully. Note that if the programmer attempts to lock a previously locked line, the line is re-loaded and locked as normal.

This function can be used to lock time critical code such as interrupt routines in the cache. This means that the performance is not dependent on the speed of the memory system. The SB flag in `IC_CTRL` can be checked to verify the completion of a `IC_LIL` operation.

When SecureShield 2+2 mode is configured

Table 22-6 IC_LIL Description

Operating Mode	Behavior
Core or debugger is in the normal mode	■ If the cache line is a hit (already present in the cache) and the line is marked as present in the normal memory space of the MPU lookup, the line is locked and the success bit is set. ■ If the cache line is a hit and the line is marked as present in the secure memory space of the MPU lookup, the line locking fails, no operating is executed on the cache line, and the success bit is cleared. ■ If the cache line is a miss, and the line is marked as present in the secure memory space of the MPU lookup, the line locking fails, no operating is executed on the cache line, and the success bit is cleared. ■ If the cache line is a miss and the line is marked as present in the normal memory space of the MPU lookup, and all the cache ways are all valid secure or valid locked, the lock fails (all cache way are locked, the cache line is not locked), and the success bit is cleared. ■ If the cache line is a miss, and the line is marked as present in the normal memory space of the MPU lookup: if there is a valid unlocked normal cache way or invalid cache way, the cache line is refilled and the dirty cache line is flushed back to the memory if appropriate. The cache line is then marked as present in the normal memory space and the success bit is set.
Core or debugger is in the secure mode	■ Writing a system memory address to IC_LIL causes the cache to load the appropriate cache line. The line is then locked so it cannot be overwritten when a cache miss occurs. If the line cannot be locked in the cache because all of the ways in the set that the address maps to are already locked, the SB flag in IC_CTRL is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when an IC_LIL operation completed successfully. Note that if the programmer attempts to lock a previously locked line, the line is re-loaded and locked as normal.



Note

If all lines in a set are locked, the cache performs the requested access direct with main memory.

Line locking using addresses that are mapped to other L1 memory components, such as the ICCM, do successfully lock the line. However, any subsequent instruction fetches still return data from the memory component that has priority, in this case the ICCM.



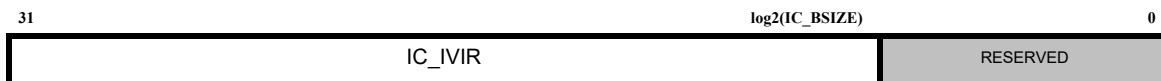
Caution

Be aware that locking all the ways of a set prevents any further areas of memory from being cached if they map to the same set. Line-locking uncacheable regions might result in unpredictable memory accesses.

22.2.4 Invalidate Instruction-Cache Start Region, IC_IVIR

Address: 0x16
 Access: W
 Reset: 0x0000_0000

Figure 22-4 IC_IVIR Register



For non-aliased configurations, the IC_IVIR register provides the full physical address (including the tag and index fields) of the region start and end addresses of the region to be invalidated. The starting address of the region is programmed in the upper bits of IC_IVIR, starting at bit $\log_2(\text{IC\_BSIZE})$. IC_BSIZE is the configured instruction cache line size. When the processor is subject to VIPT aliasing in the instruction cache, the IC_PTAG register is also used in region operations. The region invalidate operation combines the PPN in IC_PTAG with the set indexes in IC_IVIR and IC_ENDR. No other bits in IC_IVIR and IC_ENDR registers are used. There is a single IC_PTAG register that provides the physical page number for both start and end of region. Therefore region invalidate cannot span a page boundary.

When a system memory address is written to this register, the cache region starting with the cache line that maps to the given address (IC_IVIR), including locked lines, is invalidated. The operation invalidates all lines matching addresses from the given address to the address previously written to the IC_ENDR (end region) address. The address specified by IC_ENDR is excluded. You must set the IC_ENDR register before writing to the IC_IVIR register.

The SB flag in IC_CTRL can be checked to verify the completion of an IC_IVIR operation. For the behavior of the SB flag for this operation, see the IC_CTRL.SB bit description.

22.2.5 Invalidate Instruction-Cache End Region, IC_ENDR

Address: 0x17

Access: RW

Reset: 0x0000_0000

Figure 22-5 IC_ENDR Register



For non-aliased configurations, the IC_IVIR register provides the full physical address (including the tag and index fields) of the region start and end addresses of the region to be invalidated.

For aliased configurations (cache way size is larger than the MMU page size), this register holds only the index bits of the region end address and the IC_PTAG register is used to program the physical page number (PPN) of the physical page containing the region to be invalidated.

This register is used to hold the last program memory address up to which the instruction cache is invalidated. During an invalidate operation, the cache region starting from IC_IVIR to IC_ENDR is invalidated. The address specified by IC_ENDR is excluded. You must set this register before writing to the IC_IVIR register.

Note that the programming model is dependent on the size of the cache: a 64 kByte cache (with aliasing) is programmed differently than a 32 kByte cache (without aliasing).

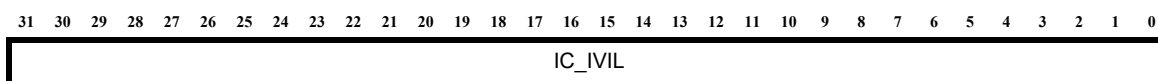
22.2.6 Invalidate Instruction-Cache Line, IC_IVIL

Address: 0x19

Access: W

Reset: 0x0000_0000

Figure 22-6 IC_IVIL Register



Typical uses of the cache invalidation are to update an interrupt vector without invalidating the whole cache, or for cache-locking maintenance. The SB flag in IC_CTRL can be checked to verify the completion of an IC_IVIL operation.

When SecureShield 2+2 mode is not configured

In non-aliased configurations, the IC_IVIL register provides the full physical address (including the tag and index used in the cache lookup) of the line to be locked or invalidated.

In aliased instruction cache configurations (cache way size is larger than the OLM page size), this register holds only the index bits of the line address and the IC_PTAG register is used to program the physical page number (PPN) of the physical page containing the line to be invalidated. The IC_PTAG register must be setup first before triggering the operation with an SR to the IC_IVIL register specifying the virtual index of the line to be invalidated. The instruction cache invalidate instructions operate on untranslated physical addresses. For aliased configurations, an SR to the IC_IVIL register (invalidate) uses the virtual index from the address written to IC_IVIL. No other bits from this register is used. The virtual index is combined with the physical page number in IC_PTAG to locate the cache line to be invalidated.

When a system memory address is written to this register the cache invalidates a cache line that maps to the given address, including locked lines. If the location specified is not in the cache, the SB flag in IC_CTRL is cleared (SB=0). Otherwise, the SB bit is set (SB=1).

When SecureShield 2+2 mode is configured

Table 22-7 IC_IVIL Description

Operating Mode	Behavior
Core or debugger is in the normal mode	When a system memory address is written to this register, the cache invalidates a cache line if the cache line address is marked as a present in the normal memory space by the MPU lookup table. The locked lines are also invalidated. The SB bit is set (SB=1) If the cache line is marked by the MPU lookup table as present in the secure memory space, the scenario is equivalent to a cache miss, and the SB flag in IC_CTRL is cleared bit is cleared. If a cache miss occurs: the SB flag in IC_CTRL is cleared (SB=0).

Table 22-7 IC_IVIL Description

Operating Mode	Behavior
Core or debugger is in the secure mode	When a system memory address is written to this register the cache invalidates a cache line that maps to the given address, including locked lines. If the location specified is not in the cache, the SB flag in IC_CTRL is cleared (SB=0). Otherwise, the SB bit is set (SB=1).



Caution

All SR accesses to the IC_IVIL register must be followed by three NOP instructions.

22.2.7 Instruction Cache External-Access Address, IC_RAM_ADDR

Address: 0x1A

Access: When SecureShield 2+2 mode is not configured: RW
When SecureShield 2+2 mode is configured: RW in the secure mode and secure debug unlocked mode

Reset: 0x0000_0000

Table 22-8 IC_RAM_ADDR for Cache Controlled RAM Access (AT==1)

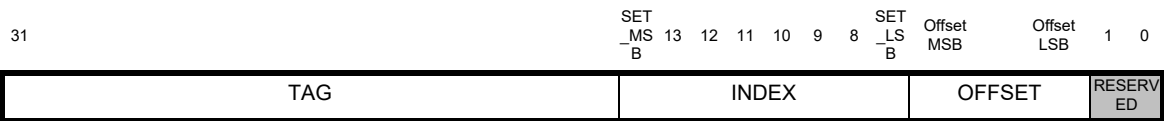


Figure 22-7 IC_RAM_ADDR Register for Direct-Cache-Access (AT==0)

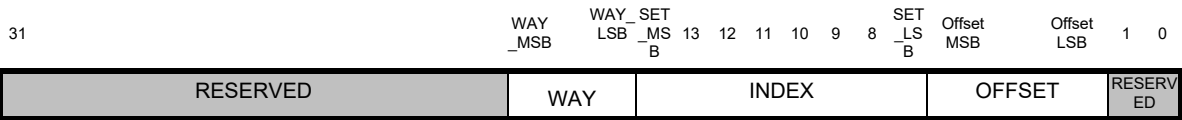
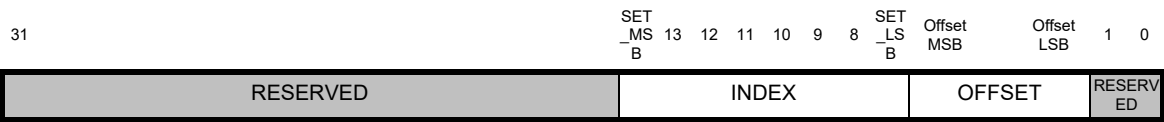


Table 22-9 IC_RAM_ADDR for Aliased Configurations



The interpretation of the register is defined by the AT (Address Debug Type) flag in the IC_CTRL register. See [Table 22-10](#) for more information.

Table 22-10 IC_RAM_ADDRESS Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the OLM page size)
Direct Mode (AT==0)	■ IC_PTAG is not used ■ SR to IC_RAM_ADDR: - triggers a read from RAM - WAY/INDEX/OFFSET specified by IC_RAM_ADDR - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM, but the index field of IC_TAG is used. IC_DATA is updated with the data value from the instruction cache data RAM ■ SR to IC_TAG updates the IC_TAG register and has the side-effect of writing the new value to the tag RAM at the location specified by the way and index field in IC_RAM_ADDR. ■ SR to IC_DATA updates the IC_DATA register and has the side-effect of writing to the new value to the instruction cache data RAM at the location specified by way, index and offset in IC_RAM_ADDR. ■ LR from IC_RAM_ADDR has no side-effect and only returns the value in that register. ■ LR from IC_TAG and IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs.	Same as non-aliased except the index field of IC_TAG is not used (remains 0).

Table 22-10 IC_RAM_ADDRESS Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the OLM page size)
Cache Controlled Mode (AT==1)	■ SR to IC_RAM_ADDR: - triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address; in this case, the physical address is specified in the IC_RAM_ADDR - Selects set and way matching the given address - updates IC_CTRL.SB with cache hit status - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM. IC_DATA is updated with the data value from the instruction cache data RAM; the index field of IC_TAG is updated with a copy from IC_RAM_ADDR. ■ SR to IC_TAG or IC_DATA triggers an instruction error exception. ■ LR from IC_TAG and IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs.	■ SR to IC_RAM_ADDR: - The IC_PTAG register must be programmed first with the physical page number. - Then the IC_RAM_ADDR must be programmed with the virtual index and offset; this triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address but tag matching uses the full PPN - updates IC_CTRL.SB with the cache hit status - If cache hit: IC_TAG gets the physical tag, lock and valid bits, but not the index. IC_DATA gets the data word from the instruction cache data RAM. IC_PTAG is unchanged ■ SR to IC_TAG or IC_DATA triggers an instruction error exception ■ LR from IC_TAG and IC_DATA only retrieves the values in those registers, but doesn't trigger an update of these registers with values from the tag and data RAMs

When the core is operating in the secure mode or the debug port is unlocked in the secure mode:

Reading from or writing to this register allows the programmer to access the valid bit, lock bit, and tag of a cache line specified by the IC_RAM_ADDR register and the AT flag found in IC_CTRL. An SR to the IC_RAM_ADDRESS register also triggers an update of the IC_TAG and IC_DATA registers.

When the core is operating in the normal mode, attempts to write to the advanced cache control registers raise a privilege violation.

When the debug port is unlocked in the normal mode:

Table 22-11 Cache Line Access in the Debug Unlocked in the Normal Mode

	Cache Line is in Secure Memory Space	Cache Line is in the Normal Memory Space
AT==0; and the cache line is a hit	The IC_TAG, IC_XTAG, and IC_DATA registers are cleared. The Success bit is cleared.	The cache line is read from RAM by Direct-access, and the IC_TAG, IC_XTAG, and IC_DATA registers are updated. The Success bit is set.
AT==1; and the cache line is a hit	The IC_TAG, IC_XTAG, and IC_DATA registers are cleared. The Success bit is cleared.	The cache line is read, and the IC_TAG, IC_XTAG, and IC_DATA registers are updated. The Success bit is set.
AT==1, and the cache line is a miss	The Success bit is cleared.	The Success bit is cleared.

22.2.8 Instruction-Cache Tag Access, IC_TAG

Address: 0x1B

Access: When SecureShield 2+2 mode is not configured: RW
When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x0000_0000

Figure 22-8 IC_TAG Register for Reads

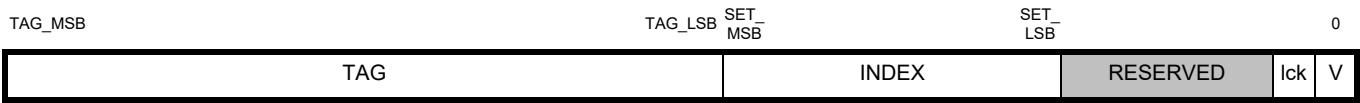


Table 22-12 IC_TAG Register for Reads in an Aliased Configuration

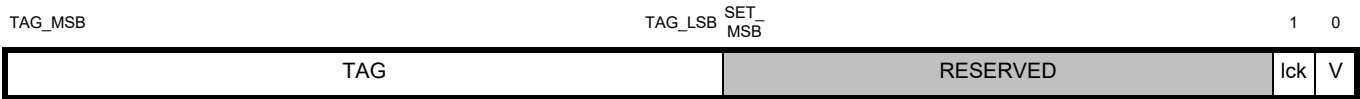
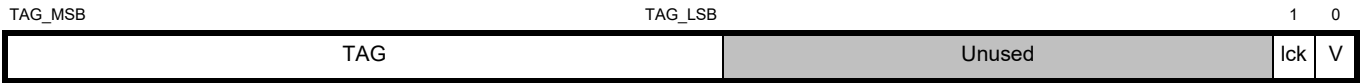


Figure 22-9 IC_TAG Register for Writes



Following are the definitions of the various fields in this register:

- SET_LSB = $\text{Log}_2(\text{IC\_BSIZE})$
- SET_MSB = $\text{Log}_2(\text{IC\_SIZE}/\text{IC\_WAYS}) - 1$
- TAG_LSB = $\text{Log}_2(\text{IC\_SIZE}/\text{IC\_WAYS})$
- TAG_MSB = $\text{PC\_SIZE} - 1$

Reading from or writing to this register allows the programmer to access the valid bit, lock bit, and tag of a cache line specified by the IC_RAM_ADDR register and the AT flag found in IC_CTRL. An SR to the IC_RAM_ADDRESS register also triggers an update of the IC_TAG and IC_DATA registers.

Table 22-13 describes the IC_TAG register behavior.

Table 22-13 IC_TAG Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the OLM page size)
Direct Mode (AT==0)	■ SR to IC_TAG updates the IC_TAG register and has the side-effect of writing the new value to the tag RAM at the location specified by the way and index field in IC_RAM_ADDR ■ LR from IC_TAG only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ IC_PTAG is not used. ■ SR to IC_RAM_ADDR: - triggers a read from RAM - WAY/INDEX/OFFSET specified by IC_RAM_ADDR - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM, but the index field of IC_TAG is used. IC_DATA is updated with the data value from the instruction cache data RAM	Same as non-aliased except the index field of IC_TAG is not used (remains 0)

Table 22-13 IC_TAG Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the OLM page size)
Cache Controlled Mode (AT==1)	■ SR to IC_TAG triggers an instruction error exception ■ LR from IC_TAG only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ SR to IC_RAM_ADDR: - triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address; in this case, the physical address is specified in the IC_RAM_ADDR - Selects set and way matching the given address - updates IC_CTRL.SB with cache hit status - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM. IC_DATA is updated with the data value from the instruction cache data RAM; the index field of IC_TAG is updated with a copy from IC_RAM_ADDR.	■ SR to IC_TAG triggers an instruction error exception ■ LR from IC_TAG only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ SR to IC_RAM_ADDR: - The IC_PTAG register must be programmed first with the physical page number. - Then the IC_RAM_ADDR must be programmed with the virtual index and offset; this triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address but tag matching uses the full PPN - updates IC_CTRL.SB with the cache hit status - If cache hit: IC_TAG gets the physical tag, lock and valid bits, but not the index. IC_DATA gets the data word from the instruction cache data RAM. IC_PTAG is unchanged

**Caution**

Take care when debugging any software that accesses the TAG RAM via the IC_TAG register.

As part of the host debugger protocol to access memory, an invalidate-cache operation is carried out after any write to memory. This invalidate cache operation updates tag RAMs and therefore modifies any value that has been written to the tag RAM by the software. An invalidate cache operation goes ahead even if the instruction cache is disabled.

Valid Bit (V[0])

The valid bit (V) indicates whether the data associated with the tag is valid or not. When V=1 the data is valid.

Lock Bit (LCK[1])

The lock bit (LCK) indicates whether the data associated with the tag is locked. The data entry is locked when LCK=1.

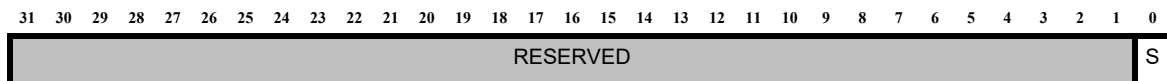
22.2.9 Instruction Cache Secure Bit Tag Register, IC_XTAG

Address: 0x1C

Access: RW in the secure mode only

Reset: 0x0000_0000

Figure 22-10 IC_XTAG Register



This register allows you to read from and write to the S bit of the instruction cache tags for direct RAM access.

A read of 1 indicates that the corresponding cache line is in the secure memory space. A read of 0 indicates that the corresponding cache line is in the normal memory space. When the software or the debugger writes to this bit, they must ensure that this bit matches the MPU protection attributes for the corresponding cache line. For example, if the cache line is in the secure region, the S bit must be written as 0.

Attempts to access this register in the normal mode raises a privilege violation exception. If the secure debugger is unlocked in the normal mode, the debugger can only read the tags that are tagged in the normal mode; attempts to write to this register in the are ignored.

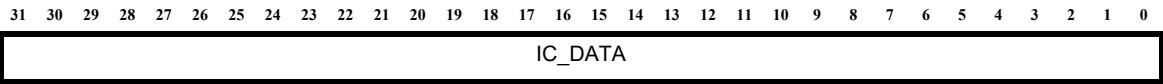
22.2.10 Instruction Cache Data Access, IC_DATA

Address: 0x1D

Access: When SecureShield 2+2 mode is not configured: RW
When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x0000_0000

Figure 22-11 IC_DATA Register



Reading from or writing to the IC_DATA register allows the programmer to access a data word from a particular cache line, using the address in IC_RAM_ADDR. An SR to the IC_RAM_ADDRESS also triggers an update of the IC_TAG and IC_DATA registers.

Table 22-14 IC_DATA Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the OLM page size)
Direct Mode (AT==0)	■ SR to IC_DATA updates the IC_DATA register and has the side-effect of writing to the new value to the instruction cache data RAM at the location specified by way, index, and offset in IC_RAM_ADDR ■ LR from IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ IC_PTAG is not used ■ SR to IC_RAM_ADDR: - triggers a read from RAM - WAY/INDEX/OFFSET specified by IC_RAM_ADDR - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM, but the index field of IC_TAG is used. IC_DATA is updated with the data value from the instruction cache data RAM	Same as non-aliased except the index field of IC_TAG is not used (remains 0)

Table 22-14 IC_DATA Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the OLM page size)
Cache Controlled Mode (AT==1)	■ SR to IC_DATA triggers an instruction error exception ■ LR from IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ SR to IC_RAM_ADDR: - triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address; in this case, the physical address is specified in the IC_RAM_ADDR - Selects set and way matching the given address - updates IC_CTRL.SB with cache hit status - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM. IC_DATA is updated with the data value from the instruction cache data RAM; the index field of IC_TAG is updated with a copy from IC_RAM_ADDR	■ SR to IC_DATA triggers an instruction error exception ■ LR from IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ SR to IC_RAM_ADDR: - The IC_PTAG register must be programmed first with the physical page number. - Then the IC_RAM_ADDR must be programmed with the virtual index and offset; this triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address but tag matching uses the full PPN - updates IC_CTRL.SB with the cache hit status - If cache hit: IC_TAG gets the physical tag, lock and valid bits, but not the index. IC_DATA gets the data word from the instruction cache data RAM. IC_PTAG is unchanged

22.2.11 Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG

Address: 0x1E
 Access: RW
 Reset: 0x0000_0000

Figure 22-12 IC_PTAG Register



When the ARCV2 processor has an instruction cache and OLM, and the instruction cache way size (depth) is larger than the smallest page size (size0), there is the possibility of cache aliasing due to the VIPT Implementation of instruction cache. In this case, the IC_PTAG register is present in the build and is used to specify the tag field of addresses for cache controlled mode operations and cache line operations (lock, invalidate). In the aliased configuration (cache way size is larger than the OLM page size), IC_PTAG [and IC_PTAG_HI, if the physical address extension is enabled] is used for the physical page number, before the virtual address (index) is written into the respective IC_* aux register.

A write to IC_PTAG sets the value of the IC_PTAG register, but has no other side-effect and does not trigger a write to the TAG RAMs or other parts of the instruction cache.

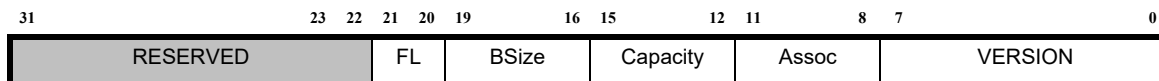
A read from IC_PTAG returns the value of IC_PTAG, but has no other side-effect and does not trigger a read from the TAG RAMs or other parts of the instruction cache.

22.2.12 Instruction Cache Configuration Register, I_CACHE_BUILD

Address: 0x77

Access: R

Figure 22-13 I_CACHE_BUILD Register



The Instruction Cache Build register (I_CACHE_BUILD) indicates the version number of the first-level private instruction cache of the processor, along with the configuration of the cache itself.

Table 22-15 I_CACHE_BUILD Field Descriptions

Field Name	Bit	Description
VERSION	[7:0]	Version number 0x0: No I_CACHE_BUILD register 0x1: Reserved 0x2: ARCompact V1, fixed 32-byte line size 0x3: ARCompact V1, variable line size 0x4: ARCV2 0x5: current version. Added support for the secure bit in the cache tags. 0x7: Support for region-wise cache operations <i>All other values are Reserved</i>
Assoc	[11:8]	Cache Associativity 0x0: Direct-mapped (1-way set associative) 0x1: Two-way set associative 0x2: Four-way set associative 0x3: Eight-way set associative <i>All other values are Reserved</i>

Table 22-15 I_CACHE_BUILD Field Descriptions (Continued)

Field Name	Bit	Description
Capacity	[15:12]	Cache capacity ■ 0x0: Reserved ■ 0x2: 2 Kbytes ■ 0x3: 4 Kbytes ■ 0x4: 8 Kbytes ■ 0x5: 16 Kbytes ■ 0x6: 32 Kbytes ■ 0x7: 64 Kbytes <i>All other values are Reserved</i>
BSize	[19:16]	Block Size, indicates the cache block size in bytes. ■ 0x0: 8 bytes ■ 0x1: 16 bytes ■ 0x2: 32 bytes ■ 0x3: 64 bytes <i>All other values are Reserved</i>
FL	[21:20]	Feature Level, indicates locking and debug feature level ■ 0x0: Basic cache, with cache invalidation but no locking or debug features ■ 0x1: Line lock and invalidate features also supported ■ 0x2: Line lock, invalidate, and advanced debug features also supported ■ 0x3: Reserved

22.3 Exceptions

See [Table 6-6](#) on page 268.

23

DCCM

23.1 DCCM Auxiliary Registers

This section describes the ARCV2 Data Closely Coupled Memories (DCCM). DCCM is designed to hold performance-critical data.

The ARCV2 architecture includes the following auxiliary registers and build configuration registers when the DCCM is enabled.

Table 23-1 DCCM Auxiliary Registers

Address	Auxiliary Register Name	Description
0x18	DCCM Base Address, AUX_DCCM	Start address of the DCCM
0x74	DCCM Configuration Register, DCCM_BUILD	DCCM build configuration register

23.1.1 DCCM Base Address, AUX_DCCM

Address: 0x18

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Default 0x8

Figure 23-1 AUX_DCCM, base address for DCCM (ADDR_SIZE == 32)



Figure 23-2 AUX_DCCM, base address for DCCM (ADDR_SIZE == 16)

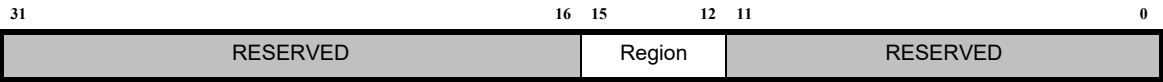


Figure 23-3 AUX_DCCM, base address for DCCM (ADDR_SIZE == m)



When a DCCM is configured in a processor, the AUX_DCCM register identifies the region in which the DCCM is contained. DCCM can be mapped to a memory region in the range 8 to 15.

If DCCM is not configured, this register does not exist. The most-significant relevant bit of AUX_DCCM is determined by the configured size of the address bus, ADDR_SIZE, such that the 4-bit Region number for the DCCM is always located at positions [ADDR_SIZE-1:ADDR_SIZE-4].

In systems with both DCCM and external memory, these memories may overlap. When reading from an address at which such an overlap occurs, the DCCM is always accessed in preference to external memory. Repositioning the DCCM does not affect the contents of external memory. Similarly, external DMA accesses to addresses that overlap external memory and DCCM do not affect the contents of DCCM.

You can configure the DCCM region using the ARChitect tool; on reset, the Region field is initialized to this value.

23.1.2 DCCM Configuration Register, DCCM_BUILD

Address: 0x74

Access: R

The DCCM RAM Configuration register (DCCM_BUILD) describes the size and version number of the Data Closely Coupled Memory.

Figure 23-4 DCCM_BUILD Register

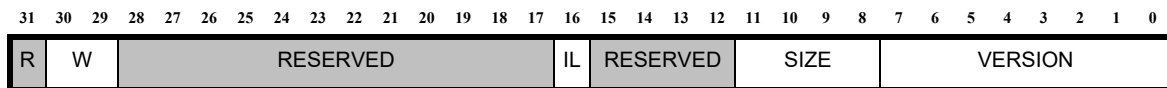


Table 23-2 DCCM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	0x4: DCCM without secure 2+2 mode configured 0x5: DCCM with secure 2+2 mode configured. 0x7: Supports wide DCCM
SIZE0	[11:8]	Size of DCCM RAM 0x0: Not present 0x1: 512B 0x2: 1KB 0x3: 2KB 0x4: 4KB 0x5: 8KB 0x6: 16KB 0x7: 32KB 0x8: 64KB 0x9: 128KB 0xA: 256KB 0xB: 512KB 0xC: 1MB 0xD = 2 MB
IL	[16]	Indicates if DCCM supports interleaving 1: DCCM supports interleaving; DCCM is split into even and odd instances to enable single cycle unaligned accesses 0: DCCM does not support interleaving

Table 23-2 DCCM_BUILD Field Descriptions (Continued)

Field	Bit	Description
W (DCCM_ WORD_ WIDTH)	[30:29]	■ 0x0: 32bit ■ 0x1: 64 bit ■ 0x2: 128 bit ■ 0x3: Reserved

23.1.3 Exceptions

See [Table 6-6](#) on page 268.

24

Data Cache

The ARCV2 architecture includes additional auxiliary registers when the data cache component is enabled. The number of additional auxiliary registers depend on the chosen `-dc_feature_level` configuration parameter.

The data cache contains advanced debug facilities that allow the programmer to interrogate and control the data cache RAM contents.

The data cache allows the programmer to obtain direct access to the internal cache RAMs through either the debug interface bus or through LR and SR instructions. This capability provides an invaluable debug environment when using the data cache.

24.1 Data Cache Auxiliary Registers

Table 24-1 Basic Data Cache Auxiliary Registers (DC_FEATURE_LEVEL==0)

Address	Auxiliary Register Name	Description
0x47	Invalidate Data Cache, DC_IVDC	Invalidate cache
0x48	Data Cache Control Register, DC_CTRL	Data cache control register
0x4B	Flush Data Cache, DC_FLSH	Flush data cache
0x5D	Non-cached Memory Region, AUX_CACHE_LIMIT	Start address of the first non-cached data region

Table 24-2 Data Cache Lock/Flush Registers (DC_FEATURE_LEVEL > 0)

Address	Auxiliary Register Name	Description
0x49	Lock Data Cache Line, DC_LDL	Lock data line
0x4A	Invalidate Data Line, DC_IVDL	Invalidate data line
0x4C	Flush Data Line, DC_FLDL	Flush data line
0x4D	Data-Cache Region Start Address, DC_STARTR	Data cache region operation start address

Table 24-2 Data Cache Lock/Flush Registers (DC_FEATURE_LEVEL > 0) (Continued)

Address	Auxiliary Register Name	Description
0x4E	Data-Cache Region End Address, DC_ENDR	Data cache region operation end address

Table 24-3 Data Cache Auxiliary Registers for Direct RAM Access (DC_FEATURE_LEVEL >1)

Address	Register Name	Description	Initial Value
0x58	Data Cache External Access Address, DC_RAM_ADDR	Data cache external access RAM address	0x0000 0000
0x59	Data Cache Tag Access, DC_TAG	Data cache tag access	0x0000 0000
0x5A	Data Cache Secure Bit Register, DC_XTAG	Indicates the secure mode of a cache line in the data cache tag	
0x5B	Data Cache Data Access, DC_DATA	Data cache data access	Undefined

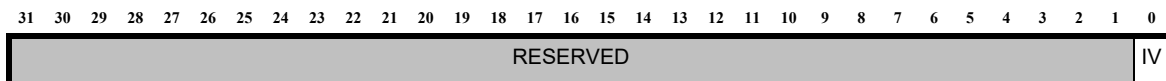
24.1.1 Invalidate Data Cache, DC_IVDC

Address: 0x47

Access: W

Reset: 0x0000_0000

Figure 24-1 DC_IVDC Register



DC_IVDC is typically used to reset the data-cache contents upon a software process switch.

Table 24-4 DC_IVDC Field Descriptions

Flag Name	Bit	Reset	Description	Access Type
IV	[0]	0	Invalidate Data Cache: Invalidates entire data cache 0x0: No Action 0x1: Invalidate Data Cache	W
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.	W

When SecureShield 2+2 mode is not configured:

Writing a 1 to the IV flag (bit 0) in the DC_IVDC register invalidates and unlocks the entire data cache.

Upon a write to the IV flag, the operation of the invalidate function is determined by the Invalidate Mode (IM) flag that is found in DC_CTRL (bit 6).

All cache lines are invalidated and unlocked, and each line entry is processed according to the state of the IM flag.

The IM flag (bit 6, DC_CTRL) controls whether a cache line entry that is marked as being dirty is flushed, invalidated, and unlocked or simply invalidated and unlocked when a DC_IVDC command is issued.

- IM = 0 (Invalidate Only) — Any write to the DC_IVDC register invalidates all cache-line entries.
- IM = 1 (Invalidate and Flush Dirty Entries) — Any write to the DC_IVDC register flushes all dirty lines and invalidates the entire data cache. The programmer must be aware that in many cases, setting the IM flag to 1 and initiating a DC_IVDC command results in an indeterminate cache-cycle time stall. The cycle-time latency is dependent on the number of entries that are dirty and the efficiency of the memory subsystem.



After triggering the cache operation (full, line or region) software needs to wait for operation to complete by polling DC_CTRL.FS bit since the operation could potentially take many cycles, depending on the number of valid or dirty lines in the cache.

When SecureShield 2+2 mode is configured

Table 24-5 DC_IVDC Description When SecureShield 2+2 Option is Configured

Operating Mode	Behavior
Core or debugger is in the normal mode	- When DC_CTRL.IM==0, the invalidate entire cache operation is ignored. - When DC_CTRL.IM==1, the normal (non-secure) data cache lines are invalidated and flushed.
Core or debugger is in the secure mode	Writing a 1 to the IV flag (bit 0) in the DC_IVDC register invalidates and unlocks the entire data cache. Upon a write to the IV flag, the operation of the invalidate function is determined by the Invalidate Mode (IM) flag that is found in DC_CTRL (bit 6). All cache lines are invalidated and unlocked, and each line entry is processed according to the state of the IM flag. The IM flag (bit 6, DC_CTRL) controls whether a cache line entry that is marked as being dirty is flushed, invalidated, and unlocked or simply invalidated and unlocked when a DC_IVDC command is issued. - IM = 0 (Invalidate Only) — Any write to the DC_IVDC register invalidates all cache-line entries. - IM = 1 (Invalidate and Flush Dirty Entries) — Any write to the DC_IVDC register flushes all dirty lines and invalidates the entire data cache. The programmer must be aware that in many cases, setting the IM flag to 1 and initiating a DC_IVDC command results in an indeterminate cache-cycle time stall. The cycle-time latency is dependent on the number of entries that are dirty and the efficiency of the memory subsystem.

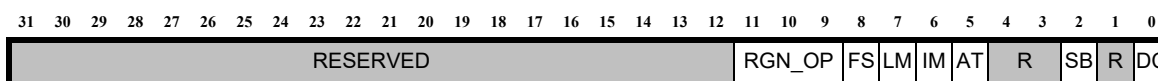
24.1.2 Data Cache Control Register, DC_CTRL

Address: 0x48

Access: RW

Reset: 0x0000_00c0

Figure 24-2 DC_CTRL Register



The control flags of the DC_CTRL are made up of DC (bit 0), AT (bit 5), IM (bit 6), and LM (bit 7). The status flags are the SB flag (bit 3) and FS flag (bit 8).

Table 24-6 DC_CTRL Control Flags

Flag Name	Bit	Reset	Description	Access Type
DC	[0]	0	Disable Cache: Enables/Disables the cache 0 – Enable Cache 1 – Disable Cache	R/W
RESERVED	[1]	0	Reserved. Read-as-zero; ignored on write.	R/W
SB	[2]	0	Success Bit: Success of last cache operation SB is for compatibility with L1 instruction or data cache. SB is set (SB=1) only for the following aux operations: - Line lock: if the line cannot be locked because all ways in the sets that the address maps to are already locked, the SB flag is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when the line lock operation completes successfully. - invalidates line or region and flushes dirty entry: if the system memory address is present in the cache, the cache line is invalidated and unlocked; if the line is dirty and IM=1, it is flushed to the memory subsystem. When the line is present, the SB flag is set (SB=1). If the line is not present, the SB=0. - flush line or region: if the system memory address is present in the cache, the SB flag is set (SB=1) upon completion regardless of whether the dirty bit is present. Cache flush does not set the SB bit.	R/W
RESERVED	[4:3]	0	Reserved. Read-as-zero; ignored on write.	R/W
AT	[5]	0	Address Debug Type: Used for debug purposes for when accessing cache RAMs. 0 – Direct Cache RAM Access 1 – Cache Controlled RAM Access	R/W
IM	[6]	1	Invalidate Mode: Selects the invalidate type 0 – Invalidate data cache only 1 – Invalidate data cache and flush dirty entries; this operation is also referred to as a purge operation.	R/W

Table 24-6 DC_CTRL Control Flags

LM	[7]	1	Lock Mode: Selects the effect of a flush command on a locked entry 0 – Disable flush on locked entry 1 – Enable flush on locked entry	R/W
FS	[8]	0	Flush Status: Status of the data-cache flush mechanism 0: Idle 1 – Flush operation in progress	R
RGN_OP	[11:9]	0	Region Operation. Selects the operation to be performed on the address region specified by the DC_STARTR and DC_ENDR registers. This register and the DC_ENDR register must be set up first before writing DC_STARTR with the region start address which triggers the operation. ■ 0x0: Region flush ■ 0x1: Region invalidate ■ 0x2 ~ 0x7: Reserved	R/W
RESERVED	[31:12]	0	Reserved. Read-as-zero; ignored on write.	R/W

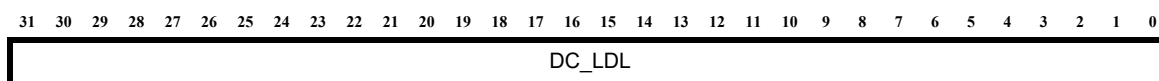
24.1.3 Lock Data Cache Line, DC_LDL

Address: 0x49

Access: W

Reset: 0x0000_0000

Figure 24-3 DC_LDL Register



When SecureShield 2+2 mode is not configured

Writing a system memory address to this register locks the cache line that maps to the given address.

- the lock data cache line operation allows any cache line in the data cache to be locked by the programmer. This allows the cache to be used as a high-speed data RAM that has a granularity of the cache-line size. This function can be used to lock critical data such as lookup tables in the cache. This means that the performance is never dependent on the speed of the memory system. The SB flag and FS flag in DC_CTRL can be checked to verify the completion of a DC_LDL operation.
- When the address is presented to DC_LDL, the cache checks to see if the requested line is present in the cache. If the line is in the cache and it is dirty, the LM flag found in DC_CTRL is checked. If LM=1, the line is flushed to memory and subsequently locked. If LM=0, the line is simply locked.
- In the event that the requested line is present in the cache but the line is not dirty, the cache line is re-fetched from memory and is used to replace the original line. This refresh mechanism ensures that the line being locked contains the most up-to-date data from memory.
- If the line that is being locked is not present in the data cache, it is simply fetched from memory and placed into the next available line. It is then subsequently locked.

When SecureShield 2+2 mode option is configured

Table 24-7 DC_LDL Description When SecureShield 2+2 mode Option is Configured

Operating Mode	Behavior
Core or debugger is in the normal mode	■ If the cache line is a hit (already present in the cache) and the line is marked as present in the normal memory space of the MPU lookup, the line is locked and the success bit is set. ■ If the cache line is a hit and the line is marked as present in the secure memory space of the MPU lookup, the line locking fails, no operating is executed on the cache line, and the success bit is cleared. ■ If the cache line is a miss, and the line is marked as present in the secure memory space of the MPU lookup, the line locking fails, no operating is executed on the cache line, and the success bit is cleared. ■ If the cache line is a miss and the line is marked as present in the normal memory space of the MPU lookup, and all the cache ways are all valid secure or valid locked, the lock fails (all cache way are locked, the cache line is not locked), and the success bit is cleared. ■ If the cache line is a miss, and the line is marked as present in the normal memory space of the MPU lookup: if there is a valid unlocked normal cache way or invalid cache way, the cache line is refilled and the dirty cache line is flushed back to the memory if appropriate. The cache line is then marked as present in the normal memory space and the success bit is set.
Core or debugger is in the secure mode	■ The lock data cache line operation allows any cache line in the data cache to be locked by the programmer. This allows the cache to be used as a high-speed data RAM that has a granularity of the cache-line size. This function can be used to lock critical data such as lookup tables in the cache. This means that the performance is never dependent on the speed of the memory system. The SB flag and FS flag in DC_CTRL can be checked to verify the completion of a DC_LDL operation. ■ When the address is presented to DC_LDL, the cache checks to see if the requested line is present in the cache. If the line is in the cache and it is dirty, the LM flag found in DC_CTRL is checked. If LM=1, the line is flushed to memory and subsequently locked. If LM=0, the line is simply locked. ■ In the event that the requested line is present in the cache but the line is not dirty, the cache line is re-fetched from memory and is used to replace the original line. This refresh mechanism ensures that the line being locked contains the most up-to-date data from memory. ■ If the line that is being locked is not present in the data cache, it is simply fetched from memory and placed into the next available line. It is then subsequently locked. ■ After a line is locked, it cannot be replaced by a cache miss. If the line cannot be locked because all of the ways in the set that the address maps to are already locked, the SB flag in DC_CTRL is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when a DC_LDL operation completes successfully.

**Caution**

The programmer must be aware that locking all the ways of a set prevents any further areas of memory from being cached if they map to the same set. An access to such a memory location goes directly to external memory.

The `st.di` and `ld.di` instructions always access external memory, even if the same address is locked in the cache.

**Note**

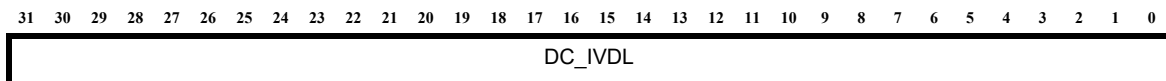
If an attempt is made to lock a previously locked line, the line is re-fetched from main memory and locked as normal.

Locked cache lines can be unlocked using the `DC_IVDL` register. The `SB` flag is used to indicate a successful lock-line operation. However, if a `DC_LDL` causes a line to be flushed to memory, it is important that the `FS` flag in `DC_CTRL` be checked along with the `SB` flag to ensure that memory is coherent. Perform this check whenever memory coherency is an issue.

24.1.4 Invalidate Data Line, DC_IVDL

Address: 0x4A
 Access: W
 Reset: 0x0000_0000

Figure 24-4 DC_IVDL Register



Typical uses are to update a data entry within a table without invalidating the entire cache or use in cache-locking maintenance (such as removing cache-address mappings from the cache). Using this function also aids the update and maintenance of shared memory architectures where addresses within a memory space are shared between multiple logic blocks (such as other processors or peripherals).

When SecureShield 2+2 mode option is not configured

When a system-memory address is written to this register, the cache invalidates the line that maps to the given address. If the line is dirty, it might be flushed before invalidation, depending on the IM bit. Locked lines are unlocked and invalidated just like unlocked lines.

The invalidate-data-line (DC_IVDL) operation allows a cache line to be removed from the cache. The method used to invalidate is selected via the IM flag (bit 6 of DC_CTRL).

- **IM = 0 (Invalidate Only)** — If the system memory address written to DC_IVDL is present in the cache, the cache line is invalidated and unlocked (if locked). For information about the SB flag for this operation, see the DC_CTRL.SB bit description.
- **IM = 1 (Invalidate and Flush Dirty Entry, also called as a purge operation)** — If the system memory address written to DC_IVDL is present in the cache, the cache line is invalidated and unlocked (if locked), and if the line is dirty it is flushed to the memory subsystem. For information about the SB flag for this operation, see the DC_CTRL.SB bit description.

When SecureShield 2+2 mode option is configured

Table 24-8 DC_IVDL Description When SecureShield 2+2 mode Option is Configured

Operating Mode	Behavior
Core or debugger is in the normal mode	■ IM = 0 (Invalidate Only) — If the system memory address written to DC_IVDL is present in the cache and the cache line is marked as present in the normal memory space based on the MPU lookup, the cache line is invalidated and unlocked (if locked), and the SB flag is set (SB=1, bit 2 DC_CTRL). If the cache line is marked as present in the secure memory space, this scenario is equivalent to a cache miss, and no operation is performed on the cache line. ■ IM = 1 (Invalidate and Flush Dirty Entry) — If the system memory address written to DC_IVDL is present in the cache and the cache line is marked as present in the normal memory space based on the MPU lookup, the cache line is invalidated and unlocked (if locked), and if the line is dirty it is flushed to the memory subsystem. Upon completion, the SB flag is set (SB=1, bit 2 DC_CTRL). If the cache line is marked as present in the secure memory space, this scenario is equivalent to a cache miss, and no operation is performed on the cache line.
Core or debugger is in the secure mode	When a system-memory address is written to this register, the cache invalidates the line that maps to the given address. If the line is dirty, it might be flushed before invalidation, depending on the IM bit. Locked lines are unlocked and invalidated just like unlocked lines. The invalidate-data-line (DC_IVDL) operation allows a cache line to be removed from the cache. The method used to invalidate is selected via the IM flag (bit 6 of DC_CTRL). ■ IM = 0 (Invalidate Only) — If the system memory address written to DC_IVDL is present in the cache, the cache line is invalidated and unlocked (if locked), and the SB flag is set (SB=1, bit 2 DC_CTRL). ■ IM = 1 (Invalidate and Flush Dirty Entry) — If the system memory address written to DC_IVDL is present in the cache, the cache line is invalidated and unlocked (if locked), and if the line is dirty it is flushed to the memory subsystem. Upon completion, the SB flag is set (SB=1, bit 2 DC_CTRL).

24.1.5 Flush Data Cache, DC_FLSH

Address: 0x4B

Access: W

Reset: 0x0000_0000

Figure 24-5 DC_FLSH Register

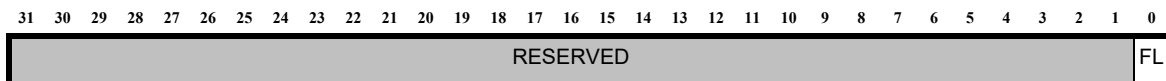


Table 24-9 DC_FLSH Control Flags

Flag Name	Bit	Reset	Description	Access Type
FL	[0]	0	Flush Data Cache:- Flush entire data cache 0x0: No Action 0x1: Flush Data Cache	W
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.	W

The DC_FLSH function is used to update main memory with the most recent copy of the data-cache contents. This is very useful for shared-memory architectures, where other processors require updated memory contents.

When SecureShield 2+2 mode is not configured

Writing a 1 to the FL flag in the DC_FLSH register initiates a flush sequence for the entire data cache. All cache sets are checked sequentially for dirty entries, and entries that contain modified data is flushed to the memory subsystem. Upon completion of a flush sequence the line-dirty bit is cleared.

The operation of a flush command on a locked entry is selected by the LM flag (bit 7) in DC_CTRL. Two sequences are possible with a locked entry.

- **LM = 0 (Disable Flush on Locked Entry)** — Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. A locked cache-line that contains a dirty entry is not flushed, keeping the cache line completely locked down.
- **LM = 1 (Enable Flush on Locked Entry)** — Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. A locked cache line that contains a dirty entry is also flushed irrespective of its locked status.

When SecureShield 2+2 mode is configured

Table 24-10 DC_FLSH Description When the Enhanced Security Package is Configured

Operating Mode	Behavior
Core or debugger is in the normal mode	■ LM = 0 (Disable Flush on Locked Entry) — Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. However, only the cache lines that are present in the normal memory space based on the MPU lookup table are flushed. A locked cache-line that contains a dirty entry is not flushed, keeping the cache line completely locked down. ■ LM = 1 (Enable Flush on Locked Entry) — Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. However, only the cache lines that are present in the normal memory space based on the MPU lookup table are flushed. A locked cache line that contains a dirty entry is also flushed irrespective of its locked status.
Core or debugger is in the secure mode	Writing a 1 to the FL flag in the DC_FLSH register initiates a flush sequence for the entire data cache. All cache sets are checked sequentially for dirty entries, and entries that contain modified data is flushed to the memory subsystem. Upon completion of a flush sequence the line-dirty bit is cleared. The operation of a flush command on a locked entry is selected by the LM flag (bit 7) in DC_CTRL. Two sequences are possible with a locked entry. ■ LM = 0 (Disable Flush on Locked Entry) — Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. A locked cache-line that contains a dirty entry is not flushed, keeping the cache line completely locked down. ■ LM = 1 (Enable Flush on Locked Entry) — Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. A locked cache line that contains a dirty entry is also flushed irrespective of its locked status.



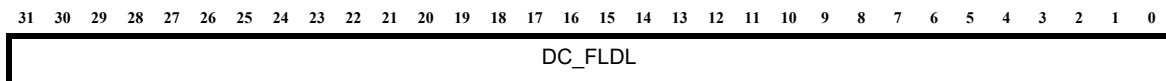
Note

Writing a 0 to the FL flag does not initiate a flush sequence.

24.1.6 Flush Data Line, DC_FLDL

Address: 0x4C
 Access: W
 Reset: 0x0000_0000

Figure 24-6 DC_FLDL Register



When SecureShield 2+2 mode option is not configured

Writing a system-memory address to DC_FLDL causes the corresponding cache-line entry to flush (if dirty). If a cache-line entry is present and is not dirty, the cache-line entry is not flushed. In the event that an entry is locked, the state of the LM flag (bit 7, DC_CTRL) controls the effect of a DC_FLDL command on the locked entry.

- **LM = 0 (Disable Flush on Locked Entry)** — A system-memory address written to the DC_FLDL register initiates a search for that entry. In the event that the requested cache line is locked, the DC_FLDL command does not generate a flush sequence, even if that entry is dirty. For information about the SB flag for this operation, see the DC_CTRL.SB bit description.
- **LM = 1 (Enable Flush on Locked Entry)** — A system-memory address written to the DC_FLDL register initiates a search for that entry. Regardless of the lock status, the DC_FLDL command generates a flush sequence if that entry is dirty. For information about the SB flag for this operation, see the DC_CTRL.SB bit description.

When SecureShield 2+2 mode option is configured

Table 24-11 DC_FLDL Description When SecureShield 2+2 mode is Configured

Operating Mode	Behavior
Core or debugger is in the normal mode	■ LM = 0 (Disable Flush on Locked Entry) — A system-memory address written to the DC_FLDL register initiates a search for that entry. If the entry is marked as present in the normal memory space by the MPU lookup table and the requested cache line is locked, the DC_FLDL command does not generate a flush sequence, even if that entry is dirty. The SB flag indicates a success when the address specified in DC_FLDL is found. If an entry is marked as present in a secure memory by the MPU lookup table, this scenario is equivalent to a cache miss, no operation is performed, and the success bit is cleared. ■ LM = 1 (Enable Flush on Locked Entry) — A system-memory address written to the DC_FLDL register initiates a search for that entry. If the entry is marked as present in the normal memory space by the MPU lookup table, regardless of the lock status the DC_FLDL command generates a flush sequence if that entry is dirty. The SB flag indicates a success (SB=1) upon completion of the flush sequence, and the dirty flag is reset (DT=0). If an entry is marked as present in a secure memory by the MPU lookup table, this scenario is equivalent to a cache miss, no operation is performed, and the success bit is cleared.
Core or debugger is in the secure mode	Writing a system-memory address to DC_FLDL causes the corresponding cache-line entry to flush (if dirty). In the event that an entry is locked, the state of the LM flag (bit 7, DC_CTRL) controls the effect of a DC_FLDL command on the locked entry. ■ LM = 0 (Disable Flush on Locked Entry) — A system-memory address written to the DC_FLDL register initiates a search for that entry. In the event that the requested cache line is locked, the DC_FLDL command does not generate a flush sequence, even if that entry is dirty. The SB flag indicates a success when the address specified in DC_FLDL is found. ■ LM = 1 (Enable Flush on Locked Entry) — A system-memory address written to the DC_FLDL register initiates a search for that entry. Regardless of the lock status, the DC_FLDL command generates a flush sequence if that entry is dirty. The SB flag indicates a success (SB=1) upon completion of the flush sequence, and the dirty flag is reset (DT=0).



Note If an entry is not dirty but is present in the cache, the SB flag indicates a success (SB=1).

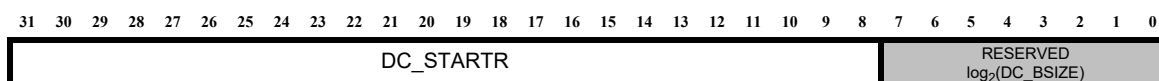
24.1.7 Data-Cache Region Start Address, DC_STARTR

Address: 0x4D

Access: W

Reset: 0x0000_0000

Figure 24-7 DC_STARTR Register



When a system memory address is written to this register, the data-cache region operation is initiated on the cache region starting with the cache line that maps to the given address (DC_STARTR), and the end address previously written to the DC_ENDR (end region) address. The address specified by DC_ENDR is excluded.

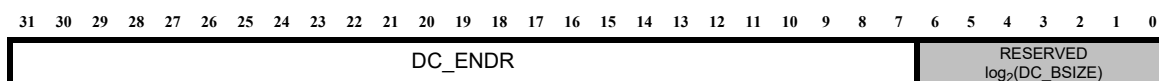
The operation performed on the cache region is defined by the DC_CTRL.RGN_OP field. You must set the DC_CTRL and DC_ENDR registers before writing to the DC_STARTR register.

This register is write-only, and writing to this register triggers the region operation. For information about the SB flag for this operation, see the DC_CTRL.SB bit description.

24.1.8 Data-Cache Region End Address, DC_ENDR

Address: 0x4E
 Access: RW
 Reset: 0x0000 0000

Figure 24-8 DC_ENDR Register



This register is used to hold the last memory address for data-cache region operations. A region operation, is performed on the cache region starting from DC_STARTR to DC_ENDR, excluding the line specified by DC_ENDR. The specific region operations to be performed are specified in by the RGN_OP field in the DC_CTRL register.

You must set this register and DC_CTRL before writing to the DC_STARTR register, which triggers the operation. The SB flag in DC_CTRL can be checked to verify the completion of an DC_STARTR operation.

The address specified should be aligned according to the cache line length, as the lower order bits in the address value are ignored. This has the effect of rounding down non-aligned addresses to the nearest cache line length boundary.

24.1.9 Data Cache External Access Address, DC_RAM_ADDR

Address: 0x58

Access: When Enhanced Security Package is not configured: RW
When Enhanced Security Package is configured: RW in the secure mode and secure debug unlocked mode

Reset: 0x0000 0000

Figure 24-9 DC_RAM_ADDR for Cache Controlled RAM Access (AT==1)

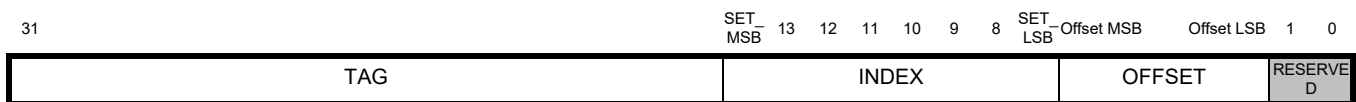
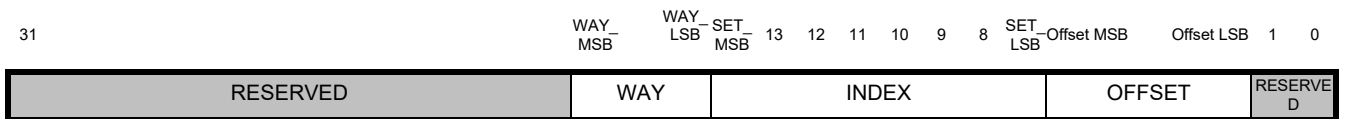


Figure 24-10 DC_RAM_ADDR Register for Direct-Cache-Access (AT==0)



This register points to a location within the cache, to be used in tandem with the DC_TAG and DC_DATA registers.

When the core is operating in the secure mode or the debug port is unlocked in the secure mode:

Reading from or writing to this register allows the programmer to access the valid bit, lock bit, and tag of a cache line specified by the DC_RAM_ADDR register and the AT flag found in DC_CTRL. An SR to the DC_RAM_ADDRESS register also triggers an update of the DC_TAG and DC_DATA registers. The read is performed according to the AT bit in the DC_CTRL register.

When the core is operating in the normal mode, attempts to write to the advanced cache control registers raise a privilege violation.

When the debug port is unlocked in the normal mode:

Table 24-12 Cache Line Access in the Debug Unlocked in the Normal Mode

	Cache Line is in Secure Memory Space	Cache Line is in the Normal Memory Space
AT==0; and the cache line is a hit	The DC_TAG, DC_XTAG, and DC_DATA registers are cleared. The Success bit is cleared.	The cache line is read from RAM by Direct-access, and the DC_TAG, DC_XTAG, and DC_DATA registers are updated. The Success bit is set.

Table 24-12 Cache Line Access in the Debug Unlocked in the Normal Mode

	Cache Line is in Secure Memory Space	Cache Line is in the Normal Memory Space
AT==1; and the cache line is a hit	The DC_TAG, DC_XTAG, and DC_DATA registers are cleared. The Success bit is cleared.	The cache line is read, and the DC_TAG, DC_XTAG, and DC_DATA registers are updated. The Success bit is set.
AT==1, and the cache line is a miss	The Success bit is cleared.	The Success bit is cleared.

This register points to a location within the cache, to be used in tandem with the DC_TAG and DC_DATA registers (refer to subsections [“Data Cache Data Access, DC_DATA”](#) and [“Data Cache Tag Access, DC_TAG”](#)).

The interpretation of the register is defined by the AT (Address Debug Type) flag in the DC_CTRL register. Writing a 0 to the AT flag in DC_CTRL selects direct cache-RAM access. Writing a 1 to the AT flag selects cache-controlled RAM access.

A detailed description of the AT flag is provided in [“Data Cache Control Register, DC_CTRL”](#) on page 919. An SR to the DC_RAM_ADDRESS register triggers a read from the data cache and updates the DC_TAG and DC_DATA registers. The read is performed according to the AT bit in the DC_CTRL register.

24.1.10 Data Cache Tag Access, DC_TAG

Address: 0x59

Access: When Enhanced Security Package is not configured: RW
When Enhanced Security Package is configured: RW in the secure mode only

Reset: 0x0000 0000

Figure 24-11 DC TAG for Reads

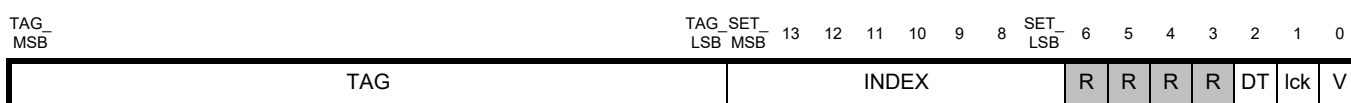


Figure 24-12 DC TAG for Writes



 Note

Following are the definitions of the various fields in this register:

- $SET\_LSB = \log_2(DC\_BSIZE)$
- $SET\_MSB = \log_2(DC\_SIZE/DC\_WAYS) - 1$
- $TAG\_LSB = \log_2(DC\_SIZE/DC\_WAYS)$
- $TAG\_MSB = ADDR\_SIZE - 1$

Reading from or writing to this register allows the programmer to access the valid bit, lock bit, dirty bit, and tag of a cache line address specified in the DC_RAM_ADDR register. In direct mode, an SR to this register updates the register and has the side-effect of writing the new value to the tag RAM at the location specified by the DC_RAM_ADDR register. A write transaction to the DC_RAM_ADDRESS also triggers an update of the DC_TAG and DC_DATA registers.

Table 24-13 DC_TAG Control Flags and Update Conditions

Flag Name	Bit	Description	Flag Update Conditions	Access Type
V	[0]	Valid Flag: Valid flag associated with the cache line 0 – Cache line not valid 1 – Cache line valid	Programmer updates to DC_TAG DC_IVDL command DC_IVDC command Cache line replacement	R/W

Table 24-13 DC_TAG Control Flags and Update Conditions (Continued)

LCK	[1]	Lock Flag: Lock flag associated with the cache line ■ 0 – Cache line not locked ■ 1 – Cache line locked	Programmer updates to DC_TAG DC_LDL command DC_IVDL command DC_IVDC command	R/W
DT	[2]	Dirty Flag: Dirty flag associated with the cache line ■ 0 – Cache line not dirty ■ 1 – Cache line dirty	Programmer updates to DC_TAG DC_FLDL command DC_FLSH command DC_IVDL command DC_IVDC command Cache line replacement	R/W

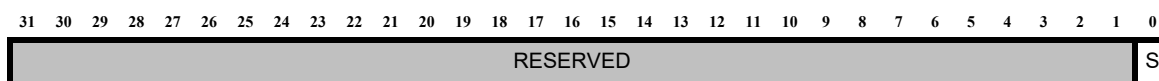
24.1.11 Data Cache Secure Bit Register, DC_XTAG

Address: 0x5A

Access: RW in the secure mode only

Reset: 0x0000_0000

Figure 24-13 DC_XTAG Register



This register allows you to read from and write to the S bit of the data cache tags.

A read of 1 indicates that the corresponding cache line is in the secure memory space. A read of 0 indicates that the corresponding cache line is in the normal memory space. When the software or the debugger writes to this bit, they must ensure that this bit matches the MPU protection attributes for the corresponding cache line. For example, if the cache line is in the secure region, the S bit must be written as 0.

Attempts to access this register in the normal mode raises a privilege violation exception. If the secure debugger is unlocked in the normal mode, the debugger can only read the tags that are tagged in the normal mode; attempts to write to this register in the are ignored.

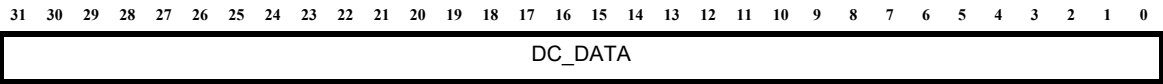
24.1.12 Data Cache Data Access, DC_DATA

Address: 0x5B

Access: When Enhanced Security Package is not configured: RW
When Enhanced Security Package is configured: RW in the secure mode only

Reset: 0x0000_0000

Figure 24-14 DC_DATA



Reading from or writing to the DC_DATA register allows the programmer to access a data from a particular cache line, using the address in DC_RAM_ADDR. In direct mode, an SR to this register updates the register and has the side-effect of writing the new value to the cache data RAM at the location specified by way, index and offset in the DC_RAM_ADDR register. An SR to the DC_RAM_ADDRESS also triggers an update of the DC_TAG and DC_DATA registers.

24.1.13 Non-cached Memory Region, AUX_CACHE_LIMIT

Figure 24-15 AUX_CACHE_LIMIT, start of non-cached regions (ADDR_SIZE == 32)

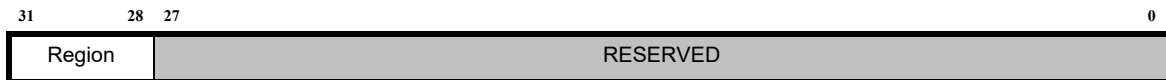


Figure 24-16 AUX_CACHE_LIMIT, (ADDR_SIZE == 16)

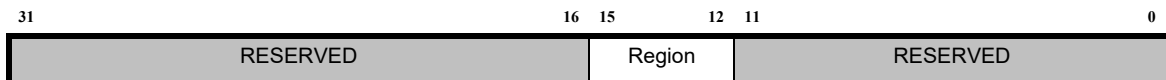


Figure 24-17 AUX_CACHE_LIMIT, start of non-cached regions (ADDR_SIZE == m)



When a data cache is attached to a processor and `dc_uncached_region == true`, the `AUX_CACHE_LIMIT` register is present and identifies the region at which the non-cached memory regions begin.

On reset, the Region field is set to 15 which is the highest region in the memory map and is always non-cacheable. To ensure a consistent view of data, the Data Cache must be flushed whenever `AUX_CACHE_LIMIT` is reduced in value.

In symmetric multi-processor implementations, this register is not replicated in each core. Therefore writes to this register affect the behavior of memory operations issued from all cores equally.

Any pending data cache control operations such as flush and invalidate must be allowed to finish before modifying this register.



Caution

Writing to this register alters the physical memory map, and may interfere with the outstanding memory operations. It is the responsibility of the software to synchronize all the outstanding memory operations before modifying this register.

24.2 Build Configuration Registers

The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

A set of baseline Build Configuration Registers are present in all ARCV2 processors. In addition, each optional ISA extension typically includes a Build Configuration Register to signify its presence and its specific configuration in each given build.

Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required

return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCV2-based system.

Table 24-14 Build Configuration Registers

Number	Name	Access	Description
0x72	Data Cache Configuration Register, D_CACHE_BUILD	R	Build configuration for: data cache

24.2.1 Data Cache Configuration Register, D_CACHE_BUILD

Address: 0x72

Access: R

Figure 24-18 D_CACHE_BUILD

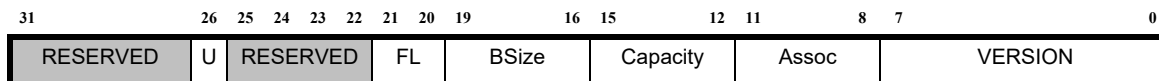


Table 24-15 D_CACHE_BUILD Field Descriptions

Field Name	Bit	Description
VERSION	[7:0]	Version number 0x0: No D_CACHE_BUILD register 0x1: reserved 0x2: ARCompact, fixed 32-byte line size 0x3: ARCompact, variable line size 0x4: ARCV2 with fixed number of cycles 0x5: Added support for the secure bit in the cache tags. 0x6: Support for region-wise cache operations <i>All other values are reserved</i>
Assoc	[11:8]	Cache Associativity 0x0: Direct-mapped (1-way set associative) 0x1: Two-way set associative 0x2: Four-way set associative 0x3: Eight-way set associative <i>All other values are reserved</i>
Capacity	[15:12]	Cache capacity 0x0: Reserved 0x2: 2 Kbytes 0x3: 4 Kbytes 0x4: 8 Kbytes 0x5: 16 Kbytes 0x6: 32 Kbytes 0x7: 64 Kbytes <i>All other values are reserved</i>

Table 24-15 D_CACHE_BUILD Field Descriptions

Field Name	Bit	Description
BSize	[19:16]	Block Size, indicates the cache block size in bytes 0x0: 16 bytes 0x1: 32 bytes 0x2: 64 bytes <i>All other values are reserved</i>
FL	[21:20]	Feature Level, indicates locking and debug feature level 0x0: Basic cache, supports cache flush and invalidation operations, but no locking or debug features 0x1: Lock, flush, and invalidation features are supported 0x2: Lock, flush, invalidation, and advanced debug features are supported 0x3: <i>reserved</i>
U	26	Indicates whether the data cache contains uncached regions. 0x0: Data cache does not include uncached regions. 0x1: Data cache includes uncached regions; the <code>AUX_CACHE_LIMIT</code> register is present and identifies the region at which the uncached memory regions begin.

24.2.2 Exceptions

See [Table 6-6](#) on page 268.

24.3 Cache Maintenance

Cache maintenance instructions provide a mechanism for flushing or purging data from an L1 cache.

There are two types of maintenance commands – *flush* and *purge*. A flush command causes modified data to be copied back from the L1 data cache to the next level below in the hierarchy. This would typically be the L2 cache, or external memory if there is no L2 cache or if the line is not already present in L2. A purge command first performs a flush operation, and then invalidates the line in L1 so that it cannot be read without triggering a subsequent L1 cache miss.

You can specify the addresses for the flushing and purging operations in the following ways:

- **Cache-based:** flush or purge the whole of the L1 data cache, regardless of its contents.
- **Region-based:** flush or purge a region of the physical address space, applying the operation only to those lines in L1 cache that lie within the specified region.

24.3.1 Cache-based Purge and Flush Operations

When coherency transactions are involved, the cache-based and region-based purging or flushing behave differently. When the whole cache is flushed/purged, the operation applies only to the L1 data cache of the core that issued the flush/purge command.

The whole-cache flush operation is useful when preparing a core for a deep sleep state, where it may be powered down. This operation ensures that any modified data in the L1 data cache is copied to memory before the L1 cache is powered down.

24.3.2 Region-based Flush

In a region-based flush operation, each cache line in the region is effectively flushed from all L1 data caches in the cluster. This operation is implemented using the coherency protocol.

The region-based flush operation is useful when forcing explicit consistency between data and instruction spaces – perhaps after self-modifying code, or when the O/S kernel executes a process.

24.3.3 Region-Based Purge

In a region-based purge operation, each cache line in the region is effectively purged from all L1 data caches in the cluster. This operation is implemented using the coherency protocol.

In region-based purge operations, any copies of the cache lines within the specified region are not retained in the L1 data caches. The `DC_CTRL.IM` bit determines the mode of operation of cache maintenance purge instructions:

- `DC_CTRL.IM = 1`: Dirty lines are copied-back before the line is invalidated.

`DC_CTRL.IM = 0`: Dirty lines are not copied-back before the line is invalidated. The initiator of the invalidate maintenance instruction invalidates the line in its L1 data cache. No copy-back is performed for this line, regardless of its state in all L1 data caches.

25

Task Switching Component

25.1 Overview

This section describes the auxiliary registers that are included in the processor when the task switch component is configured.

The task switching component includes a job translation feature that includes auxiliary registers.

Table 25-1 Task Switch Auxiliary Registers

Address	Auxiliary Register Name	Description
0x490	JLAT Segment 1 Base Address, JOB_BASE1	Base address for access to DCCM through the first JLAT segment
0x491	JLAT Segment 2 Base Address, JOB_BASE2	Base address for access to DCCM through the second JLAT segment
0xF80	Task Switch Build Register, TASK_SWITCH_BUILD	Task Switch build configuration register

25.1.1 Address Translation

Job location address translation table (JLAT) translates the logical address generated by core to physical DCCM address.

The accesses to DCCM is distributed into three segments based on address bits 27:24 (in a 32-bit address build). Access to segments 1 and 2 are translated through JLAT. Other segments are untranslated

Each job has its region within the translated segments 1 and 2. The base for job region is specified by the JOB_BASE1 register for segment 1 and JOB_BASE2 register for segment 2.

The physical address is calculated from the logical address as follows.

Addressing first JLAT segment with job base from JOB_BASE1 register:

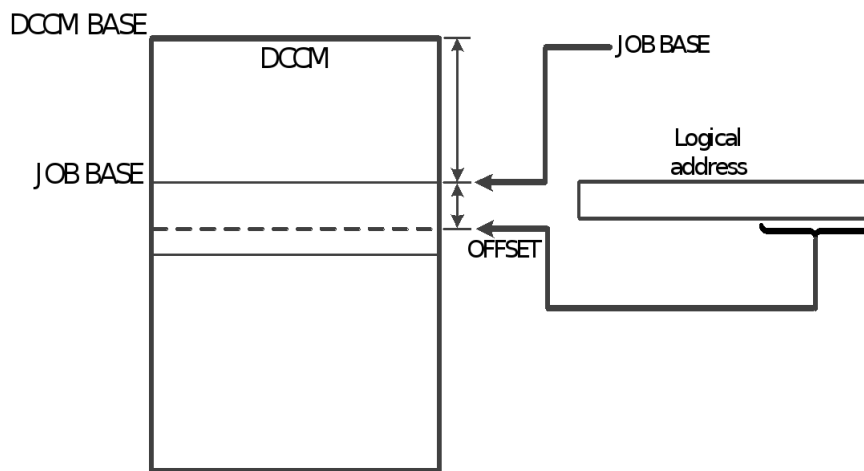
```
Physical address [31:0] = ((logical_address [31:28] == DCCM_BASE) && (logical_address
[27:24] == 1)) ?
({DCCM_BASE, 28'h0}
| {8'h0, JOB_BASE1 [23:0] }
| {8'h0, logical_address [23:0]});
```

```

If addressing second JLAT segment 2 with job base from JOB_BASE2 register:
Physical address [31:0] = ((logical_address [31:28] == DCCM_BASE) && (logical_address
[27:24] == 2)) ?
({DCCM_BASE, 28'h0}
| {8'h0, JOB_BASE2[23:0]})
| {8'h0, logical_address [23:0]}) ;

```

Figure 25-1 JLAT with JOB_BASE



The auxiliary registers JOB_BASE1 and JOB_BASE2 can be written through the Background copy interface and the core. Shadow registers of these JOB_BASE* registers are implemented in the register bank and the active values are copied to these registers.

Following are some of the restrictions when using the JLAT for DCCM:

- Unaligned accesses crossing the job region boundaries will result in undefined behavior. For example,

```

job_base2 = 0x6_0000
Logical Address = 0x82F9_FFFE

```

It is expected the first data returned would be for Logical Address 0x82F9_FFFE (that is, translated address 0x80FF_FFF0), and the second data returned would be for Logical Address 0x82FA_0000 (that is, translated address 0x82FE_0000).

However, the first data returned would be for Logical Address 0x82F9_FFFE (that is, translated address 0x80FF_FFF0) and the second data returned would be for translated address 0x8100_0000

- If JOB_BASE is specified outside the DCCM Size, the logical address is not translated

The stack checking exception will be enabled based on logical address programmed by the User and not on JLAT translated physical address.

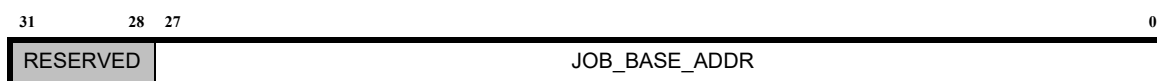
25.1.2 JLAT Segment 1 Base Address, JOB_BASE1

Address: 0x490

Access: RW

Default 0x0

Figure 25-2 JOB_BASE1 Register



This register contains the base address for access to DCCM through the first JLAT segment. This register is used in the JLAT conversion. This register is present in the build only when the **Task Switching** component is configured.

Table 25-2 JOB_BASE1 Field Descriptions

Field	Bit	Description
JOB_BASE_ADDR	[27:0]	The base address of current job in the DCCM when accessed through first JLAT segment
RESERVED	[31:28]	These bits read as zero.

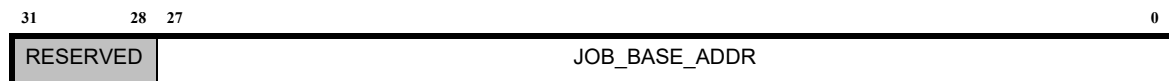
25.1.3 JLAT Segment 2 Base Address, JOB_BASE2

Address: 0x491

Access: RW

Default 0x0

Figure 25-3 JOB_BASE2 Register



This register contains the base address for access to DCCM through the second JLAT segment. This register is used in the JLAT conversion. This register is present in the build only when the **Task Switching** component is configured.

Table 25-3 JOB_BASE2 Field Descriptions

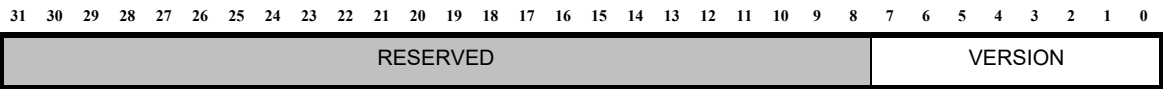
Field	Bit	Description
JOB_BASE_ADDR	[27:0]	The base address of current job in the DCCM when accessed through first JLAT segment
RESERVED	[31:28]	These bits read as zero.

25.1.4 Task Switch Build Register, TASK_SWITCH_BUILD

Address: 0xF80

Access: R

Figure 25-4 TASK_SWITCH_BUILD Register



This register indicates the presence of the task switch component and its configuration.

Table 25-4 TASK_SWITCH_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	■ 0x1: First version of the task switch component
RESERVED	[31:8]	These bits read as zero.

26

Overlay Management Unit

26.1 Overview

Overlay Management Unit (OLM) is a mechanism that provides address mapping from a single virtual address space to the physical address. The goal for the system is to achieve reasonable performance over a large code base residing in a slow secondary storage memory, but paged in as needed into faster small on-chip page RAM (PRAM). This is useful for operating environments in which virtual address aliasing is avoided in software.

26.1.1 Auxiliary Registers for Overlay Manager Unit

These auxiliary registers are provided for interaction between an operating system (or debugger) and the TLB.

Table 26-1 Special Purpose Registers for TLB Control

Auxiliary Register	Name	Description
0x45D	OLM_VA_START	Start of the translatable region
0x45E	OLM_VA_END	End of the translatable region
0x460	TLBPD0	TLB Page descriptor register 0
0x461	TLBPD1	TLB Page descriptor register 1
0x464	TLBIndex	TLB index register
0x465	TLBCommand	TLB command register (fully serializing)
0x468	TLBEnable	TLB enable (fully serializing)
0x46C	SCRATCH_DATA0	Thirty-two-bit scratch auxiliary register that can be used to store any data. The OS may for example use this register to hold the base address of the first-level page table to speed up page-table access.

26.1.2 Start of Translation Region, OLM_VA_START

Address: 0x45D

Access: RW

Reset 0x00

Figure 26-1 OLM_VA_START Register



This register determines the start of the translatable region. The translatable region includes the start address. This register only exists in configuration with the OLM component.

Programming Notes:

- Programming the start address to a region other than 0x0, 0x1, or 0x2 is ignored.
- If configured in the processor, ICCM must be 0x0 or 0x1 regions and must be in the untranslated region.
- A write to this register is serializing.

26.1.3 End of Translation Region, OLM_VA_END

Address: 0x45E

Access: R

Reset 0x8000_0000

Figure 26-2 OLM_VA_END Register



This register determines the end of the translatable region. This register only exists in configuration with the OLM component. The translatable region excludes the end address.

26.1.4 TLB Page Descriptor Register for 32-Bit Virtual Address, TLBPD0

Address: 0x460

Access: RW

Reset 0x00

Figure 26-3 Page Descriptor Format for 4KB Page Size

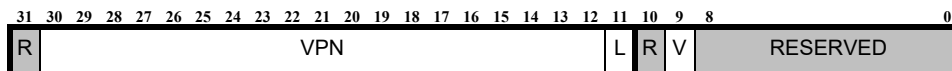


Figure 26-4 Page Descriptor Format for 8KB Page Size

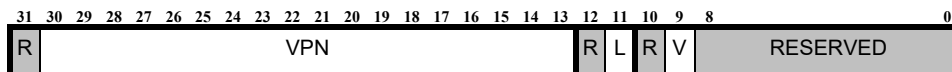
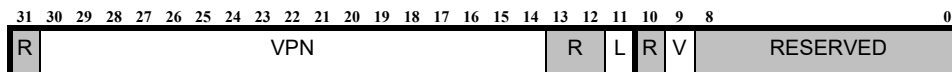


Figure 26-5 Page Descriptor Format for 16KB Page Size



A page descriptor consists of two 32-bit words; the first word relates to TLB Page Descriptor 0 (TLBPD0), the second to TLB Page Descriptor 1 (TLBPD1). Writes to this register are non-serializing.

These registers are used for the following purposes:

- To supply a page descriptor for subsequent loading into the TLB (Translation Lookaside Buffer).
- To return a page descriptor from a TLB probe operation.
- The virtual page number field is used to specify the virtual address of a TLB entry to be removed (all other fields are ignored).
- On TLB miss exceptions, the TLB Page Descriptor register 0, TLBPD0, is updated with the VPN (Virtual Page Number) associated with the address that was the cause of the TLB miss exception.

The Reserved fields are read as zero and ignored on writes.

VPN - Virtual Page Number

This field provides the virtual page number corresponding to this page descriptor. OLM checks the virtual address of an incoming request against the entries in the translation lookaside buffer. If a matching entry is found, the physical address is calculated. The field is aligned in the page descriptor to allow generation from a 32-bit address using a simple AND operation.

Bit 31 of TLB Page Descriptor 0 is not included in this field because translated memory is only available in the lower two GB of the address space.

The significant virtual page number as a function of the virtual address is calculated as follows:

$$\text{VPN} = \text{VA}[30:M] \text{ where } M \text{ is } \log_2(\text{pageSize})$$

L — Lock Bit

When this bit is set to 1, the entry is locked. The LRU algorithm does not consider a locked entry for replacement.

An entry is locked in the TLB at the time of allocation, that is, software must set the lock bit in TLBPD0 (TLBPD0.L) before writing the page in the TLB.

To unlock an entry: remove the entry (TLBDeleteEntry) and re-insert the entry.

Note:

- Entries are only locked in the normal or super-page-size TLBs. Entries in the μ TLBs (ITLB and DTLB) are not locked. The micro-TLBs also do not store the Lock bit and thus it is always read as zero (the bit is RAZ/IOW).
- When a locked entry is invalidated, the lock bit is also cleared.
- The locked entries can be inserted, deleted, or modified using the TLB commands: TLBWrite, TLBInsert, and TLBDelete.

V - Valid

This bitfield is used to indicate whether an entry in the TLB should be considered during a memory access when checking for a matching TLB entry. When cleared, it indicates that the entry slot is vacant and may be picked for replacement - invalid entries are always chosen before a valid entry is evicted.

26.1.5 TLB Page Descriptor Register for 32-Bit Physical Address, TLBPD1

Address: 0x461

Access: RW

Reset 0x00

Figure 26-6 TLB Page Descriptor Register for 4KB Page Size

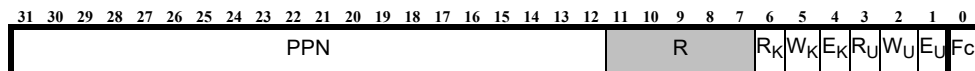


Figure 26-7 TLB Page Descriptor Register for 8KB Page Size

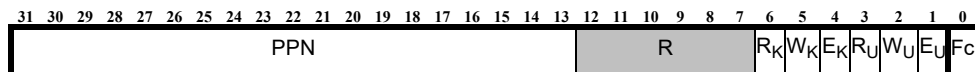
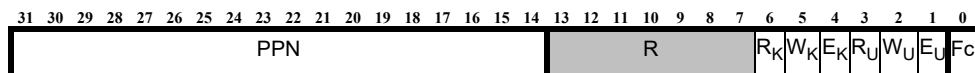


Figure 26-8 TLB Page Descriptor Register for 16KB Page Size



For a 32-bit physical address, the page descriptor consists of two 32-bit words; the first word relates to TLB Page Descriptor 0 (TLBPD0), the second to TLB Page Descriptor 1 (TLBPD1). Writes to this register are non-serializing.

PPN - Physical Page Number

This field is used for virtual-to-physical address translation, as follows:

- When the entry is marked valid:
 - This virtual memory page is present in physical memory.
 - This field describes the physical page in the main memory that is used for accesses to the virtual page. The high order bits of the physical address (page number) come from this field, and the bottom twelve bits (page offset) come from the bottom twelve bits of the virtual address.
 - The physical page can be located anywhere in the full 32-bit (4 GB) address space.
- When the entry is not marked valid:
 - This virtual page is not present in physical memory. This arrangement is only useful in an operating system supporting demand-paged virtual memory.
 - This field is typically used by the OS to indicate the location of the page in a disk-based swap file.

The physical page number of a block is the address of the block shifted right by $\log_2(\text{PageSize})$ bits (divided by *PageSize*).

R_K , W_K , E_K — Kernel Mode Permission Bits

Each page descriptor features separate access control bits for user-mode and kernel-mode tasks.

These bits do not form part of the addressing mechanism. It is not permissible to have more than one TLB entry mapped to the same virtual address, even if the access permissions do not overlap.

An exception is generated if an access is attempted that violates the access permissions for the page. The access does not complete.

These three bits control the permissions granted to tasks, interrupts, or exception handlers running in kernel mode or other operating-system functions using kernel mode. Setting a bit to true (1) indicates that the permission is granted.

- R_K — Kernel mode read permission
- W_K — Kernel mode write permission
- E_K — Kernel mode execute permission

R_U , W_U , E_U — User Mode Permission Bits

Each page descriptor features separate access control bits for user-mode and kernel-mode tasks.

Note that these bits do not form part of the addressing mechanism. It is not permissible to have more than one TLB entry mapped to the same virtual address, even if the access permissions do not overlap.

An exception is generated if an access is attempted which violates the access permissions for the page. The access does not complete.

These three bits control the permissions granted to tasks running in user mode. Setting the bit true (1) indicates that the permission is granted.

- R_U — User mode read permission
- W_U — User mode write permission
- E_U — User mode execute permission

F_C - Cached/Uncached Flag

This bit controls the data cache operation when accessing this page of system memory.

- The default is to set this bit to true to indicate that caches may be used for accesses to this page.
- When this bit is set to false, accesses to this page are sent directly to external memory, bypassing caches. This setting can be used for I/O register space or volatile data areas, such as a region of memory where data is written by a DMA transfer.

It is the responsibility of the operating system (or user code) to ensure that the cache does not contain entries from pages that are marked uncached. This requirement ensures that the cache does not contain old data that might either be flushed into the uncached page at some later point, or that might be used incorrectly if a page is subsequently marked cached.

When more than one virtual page is mapped to the same physical page, all pages must have the same setting for the cached/uncached flag.

In addition to the cached/uncached switch in the page descriptor, individual load and store instructions have a cached/uncached mode switch. Accesses are performed without caches if either the instruction or the page descriptor indicates uncached. Accesses use the caches only if both the instruction and the page descriptor indicate cached.

For data items that span a page boundary and the two relevant page descriptors' FC flags are different, an exception is generated indicating data access spanning multiple memory targets. However, If such an access uses the `.di` or direct memory (cache bypass) switch, then the memory access of both data fragments are considered uncached and thus do not result in the multiple memory targets exception.

In addition to the page descriptor FC bit and the `.di` instruction switch, there are other mechanisms that render a data access uncached. These include the DMP_PER_AUX region and AUX_CACHE_LIMIT parameters; any of these attributes individually can render the data access uncached.

26.1.6 TLB Index Register, TLBIndex

Address: 0x464

Access: RW

Reset 0x0

Figure 26-9 TLBIndex Register



This register is set by the programmer to communicate the index for TLBWrite and TLBRead commands, and set by the hardware to communicate a result from the TLBGetIndex and TLBProbe commands. Bit 31 is set to indicate an error: A value of 0x8000.0000 or above indicates an error. Writes to this register can be non-serializing. The address in the TLBIndex register is mapped as shown [Table 26-2](#).

The Reserved field is set to zero.

The following fields are described in more detail:

- [Index](#)
- [RC](#)
- [E, Error Code](#)

Index

This field is read/write.

This part of the register is set by the programmer to communicate the index for the TLBWrite and TLBRead commands, and set by the hardware to communicate a result from the TLBINSERT/TLBDELETE, TLBGetIndex, and TLBProbe commands. If an error has occurred (E is set), the index contains the error code. See [Table 26-4](#) for more information on usage.

Table 26-2 TLBIndex Addresses

Access Type	Address	Description
JTLB	0x000-0x1FF (max)	This field allows both TLBWrite and TLBRead to be performed on the JTLB RAM.
μITLB	0x1000 – 0x1003	Supports TLBRead to be performed on the μITLB.
μDTLB	0x1100 – 0x1107	Supports TLBRead to be performed on the μDTLB.

RC

This field is read-only. This three-bitfield is set by the hardware indicating results from certain commands. Writes to this field are ignored. Values other than those listed in [Table 26-3](#) are reserved.

Table 26-3 Result Code, Read Only

Code	Description
0x0	Normal operation
0x4	Duplicate TLB entries
0x7	Uncorrectable ECC error in the JTLB RAMs

E, Error Code

This field is read-only.

This bit is set by the hardware when an error has occurred or a searched entry is not found. Writes to this bit are ignored.

26.1.7 TLB Command Register, TLBCommand

Address: 0x465

Access: W

Reset: 0x00

Figure 26-10 TLBCommand Register



This register is used to initiate all transactions with the TLB. Writes to this register are serializing, that is, a pipeline flush occurs after the auxiliary write is committed. For best operating system performance, it is desirable to minimize time spent in TLB miss handlers - hence minimizing pipeline flushes is important. Data is communicated through the `TLBPD0`, `TLBPD1`, and `TLBIndex` registers. If an invalid command is written, a machine check exception is raised. Only bits [5:0] are considered in decoding command and detecting invalid command. For invalid commands the Instruction Error ECR value is 0x020000. The following commands are supported:

Table 26-4 TLB Command Register Command List

Command	Name	Description
0x1	TLBWrite	Write a TLB entry to the index location specified in <code>TLBIndex</code> and invalidate μ TLBs. This command is also used to remove entries. You can invalidate μ TLB entries by clearing their valid bits, but the entries are otherwise unchanged.
0x2	TLBRead	Read a TLB entry into <code>TLBPD0</code> or <code>TLBPD1</code> from the location specified in <code>TLBIndex</code> .
0x3	TLBGetIndex	Set <code>TLBIndex</code> to contain a suitable index location for the page descriptor in <code>TLBPD0</code> or <code>TLBPD1</code> or an error code. The <code>TLBGetIndex</code> command always returns an index where the new entry (LRU) can be loaded. The <code>TLBIndex.E</code> bit and <code>RC</code> field are also updated after execution of <code>TLBGetIndex</code> command.
0x4	TLBProbe	Searches the JTLB and if specified entry is present, returns the index of the matching entry in the <code>TLBIndex</code> register. The <code>E</code> and <code>RC</code> fields of <code>TLBIndex</code> are also updated. The page descriptor bit <code>TLBPD0[V]</code> is not considered in the search.
0x5	TLBWriteNI	Write a TLB entry to the index location specified in <code>TLBIndex</code> , without invalidating μ TLBs. Used for update on TLB miss.
0x6	IVUTLB	Invalidate μ TLBs. Used when removing entries from the page table in memory. Entries are invalidated by clearing their valid bits but the entries are otherwise unchanged.

Table 26-4 TLB Command Register Command List (Continued)

Command	Name	Description
0x7 0xA	TLBInsertEntry TLBInsertEntryJ	Write a TLB entry to the JTLB index location appropriate for the given VPN in TLBPD0 without invalidating μ TLBs. The TLBIndex register does not need to be set before issuing this command; instead, the command returns the location where the hardware placed the new entry in JTLB in TLBIndex. Note that all matching entries in all TLB sections are deleted before the new entry is inserted. The TLBPD0.V (valid) bit must be set for any entries to be considered a match. If TLBPD0.V bit is 0, then no insertion or deletion is done and the TLBIndex fields E, RC, Index are returned as 1,0x0,0x0. The matching algorithm of Insert is identical to that of delete commands and thus, delete is a subset function of insert.
0x8	TLBDeleteEntry	Invalidate entry for the page descriptor in TLBPD0, if present in the TLB (including all TLB caches: JTLB, uTLB, uDTLB). The TLBPD0.V (valid) bit must be set for any entries to be considered a match (and deleted). If TLBPD0.V bit is 0, then no insertion or deletion is done and the TLBIndex fields E, RC, Index are returned as 1,0x0,0x0.
0xB	TLBResetLRU	This command resets the LRU state and unlocks all entries within a fully associative Normal-page TLB when configured; when a memory-based 4-way JTLB is configured (256 entries or larger), the lock bits are not cleared by the TLBResetLRU command. In this case, software can update the lock bits as necessary using the TLBWrite, TLBInsert, or TLBDelete commands. After reset the LRU state is entry $N-1$ is the LRU and entry 0 is the Most Recently Used (MRU), where N is the maximum number of allocatable entries.

26.1.8 TLBWrite and TLBWriteNI

The best way to insert or delete TLB entries is to use **TLBInsertEntry/ TLBDeleteEntry** command.

The **TLBWrite** and **TLBWriteNI** commands can be used to write a TLB entry (TLBPD0/TLBPD1) to the index location specified in TLBIndex and optionally clear (invalidate) the μ TLBs.

The **TLBWrite** command invalidates μ TLBs. **TLBWriteNI** does not invalidate μ TLBs.

The operating system may determine an appropriate location for the entry by itself, or may ask the OLM hardware for a suggestion by using the **TLBGetIndex** command.

The **TLBWrite** command is also used to remove (shoot down) existing entries by loading an entry with the V-bit set false. The TLBPD0 and TLBPD1 register bits are typically set to all zeros before a **TLBWrite** command is issued. The operating system may determine the index of the entry to be removed, or may use the **TLBProbe** command to return an index that corresponds to a virtual address.

The **TLBWrite** and **TLBWriteNI** commands can still be performed when OLM is disabled (when the T-bit is 0 in the TLB enable register).

Micro-TLB entries are invalidated by clearing their valid bits but the entries are otherwise unchanged.

TLBWrite and TLBWriteNI Usage

- The page descriptor to be loaded into the TLB is brought into the TLBPD0 and TLBPD1 auxiliary registers.
- TLBIndex contains the index location to which the entry is to be loaded.
- The **TLBWrite[NI]** command code is written to the TLBCommand register.
- The TLBPD0 and TLBPD1 auxiliary registers are unchanged after the **TLBWrite** and **TLBWriteNI** operation.
- If an invalid index value is supplied (out of range), the TLB load request is ignored, and TLBIndex is loaded with an error flag E set and the result code field containing error code 0x0 (full value returned is 0x8000_0000). Note the micro-TLBs (μ ITLB and μ DTLB) are not writable and the index value for them is considered *out of range* for **TLBWrite[NI]** operations.
- Entries may be loaded with the valid bit cleared (V=0). Such entries are not considered during lookup operations; however, this feature allows an entry to be invalidated and also allows an entire save/restore of the main TLB contents to be performed.
- The **TLBWrite** command may insert, delete, or modify locked entries. The lock and valid bits are treated independently by the **TLBWrite** command. If the valid bit in PD0 (PD0.V) bit is zero and the lock bit in PD0 (PD0.L) is one, the matching entry is invalidated but its lock bit is set.

26.1.9 TLBRead

This command is used to read an entry from the TLB at the specified index location in TLBIndex and write it into TLBPD0 and TLBPD1. This command supports all valid index values, JTLB, and μ TLBs. The operating system may either determine the location to be read, or may use the **TLBProbe** command to obtain the location of an entry from a virtual address.

The **TLBRead** command operation can still be performed when OLM is disabled (when the T bit is 0 in the TLB enable register).

TLBRead Usage

- The TLBIndex register contains the location from which the entry is to be read.
- The read and write permission bits (in total of four bits) are always set to zeros when reading entries in the μ ITLB. Read and write permissions only apply to the μ DTLB.
- The execution permission bits (two bits) are always set to zeros when reading entries in the μ DTLB. Execution permission only apply to the μ ITLB.
- The reserved bits are always set to zeros when reading entries in any TLB section (writes to these bits are ignored).
- TLBPD0 and TLBPD1 registers contain the TLB entry from the specified location. Entries in the TLB that are marked as invalid are returned as they appear in the TLB.
- When a valid index is used for the read, the TLBIndex register Error flag (E) and Result Code (RC) fields are cleared.

- If an invalid index value is supplied (out of range), the TLBIndex is loaded with error flag E set and the result code field containing error code 0x0 (full value returned is 0x8000.0000), and the TLB Read operation returns an entry with all bits set to zero.
- If an uncorrectable ECC error is detected during TLBRead (or any command which reads the JTLB), the E and RC bits are set to 1, 0x7 respectively.

26.1.10 TLBGetIndex

The best way to add a new entry is to use TLBInsertEntry.

This command is used to allow the hardware to provide a TLB index to which a new entry may be loaded. This has the following benefits:

An operating system that is aware of the configuration of the TLB can implement a different or more sophisticated replacement algorithm than is supported by the hardware; however, this increases the number of cycles taken during TLB misses.

The handling of complex error conditions may be deferred to the operating system.

The **TLBGetIndex** command operation can still be performed when OLM is disabled (when the T bit is 0 in the TLB enable register).

TLBGetIndex Usage

- The page descriptor to be loaded into the TLB is written by software into the TLBPD0 and TLBPD1 registers.
 - On executing **TLBGetIndex**, the TLBIndex register is loaded with the location to which the supplied page descriptor can be loaded. JTLB uses a true LRU replacement policy.
 - When there is no uncorrectable ECC error in the JTLB PD0 RAM (if accessing JTLB), the E and RC bits are cleared on execution of TLBGetIndex, and a suitable index for the new entry is always returned in the TLBIndex.Index field.
 - If an uncorrectable ECC error is detected during TLBGetIndex (or any command that reads the JTLB), the E and RC bits are set to 1, 0x7 respectively. The index returned reflects that any cache ways with an uncorrectable error behave as if their valid bit was cleared. Thus, like other invalid way(s), they are preferred in selection as victims among the four ways; the lowest index invalid way is selected. A suitable index value for the new entry is always returned.
- Invalid ways are selected first before considering a pseudo-LRU generated victim.
 - If a getIndex command encounters an uncorrectable ECC error in the JTLB PD0 RAM (while accessing the valid bits of JTLB entries), those cache ways with an uncorrectable error behave as if their valid bit was cleared.
- This command fails when all entries are locked, that is, the error bit (TLBIndex.E) is asserted and the reason code (TLBIndex.RC) is set to zero.

26.1.11 TLBProbe

The best way to insert or delete TLB entries is to use **TLBInsertEntry**/**TLBDeleteEntry** commands.

Search the JTLB for an entry specified by the PD0 register. If an entry is present in JTLB, it returns the index of the matching entry in the TLBIndex register. If the entry is not found, the TLBIndex register E flag is set along with Result Code 0, and the index field returns 0.

The **TLBProbe** command operation can still be performed when OLM is disabled (when the T bit is 0 in the TLB enable register).

TLBProbe Usage

- The VPN[VPN_MSB:0] field of TLBPD0 contains the virtual address for which the JTLB is to be searched. The page descriptor bit V is not considered in the search.
 - Because of this command, if a single matching entry is found the TLBIndex register is loaded with the index of the matching valid entry, and the error flag and result code fields are both set to 0.
 - If no matching entry is found in the JTLB, the TLBIndex is loaded with error flag E set and the Result Code field containing error code 0x0, and the TLBIndex . Index field 0x0.
 - If more than one matching entry is found in the main TLB, the TLBIndex register is loaded with error flag E set and the Result Code field containing error code 0x4.
 - If an uncorrectable ECC error is detected during TLBProbe (or any command which reads the JTLB), the E and RC bits are set to 1 and 0x7 respectively. The index returned reflects that any cache ways with an uncorrectable error behaves as if its valid bit was cleared. Thus, if the probe hit two entries in the JTLB but one of those entries had an uncorrectable error detected, then only the other (error free) entry is declared a match; its index is returned but the E/RC code returned is 1, 0x7.
- A matching entry for the TLBProbe command is defined as a TLB entry for which:
 - The TLB entry valid (V) bit is set true.
 - Significant bits of the virtual address VPN field for JTLB section exactly match the requested (PD0) VPN.
 - If a probe command encounters an uncorrectable ECC error in the JTLB PD0 RAM (while accessing the valid bits of JTLB entries), those cache ways with uncorrectable errors behave as if their valid bit was cleared (see description above).
- Note that in version >= 0x02 of the OLM, it is possible for μ TLBs to contain entries which are not present in the main TLB. Be aware that the TLBProbe command searches entries only in the main TLB, and not the μ TLBs. TLBDeleteEntry can be used to delete a specific entry from all TLB sections. The TLBRead command can be used to examine the μ TLB entries one by one, or the IVUTLB command can be used to explicitly clear the μ TLBs when necessary.

26.1.12 IVUTLB

This command is used to allow the Operating System to explicitly clear the uTLBs when the page table in memory is changed.

This command invalidates all entries in the μ TLBs by clearing the valid bits of entries (in PD0) but does not otherwise change the entry data. Typically, when performing page table maintenance operations, the OS uses TLBDeleteEntry to remove a specific entry from all TLB sections.

In version 0x02 or later of the OLM, when TLBWriteNI or TLBInsertEntry is used by TLB miss handlers, it is possible for a page table entry to be present in one or both μ TLBs but not present in the main TLB. The TLBProbe command cannot locate the entry as it is not in the main TLB.

No parameters are required.

26.1.13 TLBInsertEntry, TLBInsertEntryJ Commands

The insert commands are used to add a new entry to the OLM TLBs; the insert commands delete the matching TLB entries in any of the TLB caches including the JTLB, μ ITLB and μ DTLB. The search for entry deletion spans all TLB caches (including the μ TLBs) regardless of the presence of the entry in the JTLB. The insert commands consider the TLBPD0 page valid in determining the scope of the search for entries to delete and the TLB where the new entry is inserted. The TLBPD0.V (valid) bit must be set for any entries to be considered a match. If TLBPD0.V bit is 0, then no insertion or deletion is done and the TLBIndex fields E, RC, Index are returned as 1, 0x0, 0x0. The matching algorithm of insert is identical to that of delete commands and thus, delete is a subset function of insert.

TLBInsertEntry is used for updates on the TLB miss.

The new entry is also forwarded to exactly one of the μ TLBs where it replaces a matching entry or an invalid entry or the LRU entry. Which μ TLB the new entry is forwarded to, depends on the execute bits; if either one or both of the execute bits is set then the entry is forwarded to μ ITLB, otherwise, to the μ DTLB. The μ TLB which is not updated with the new entry however, gets a request to invalidate any matching entry to avoid stale entries with the same VPN.

TLBInsertEntryJ: Used when many new entries are to be added to TLB. The new entry is inserted only in the JTLB and not forwarded to either μ TLB.

Both TLBInsertEntry, TLBInsertEntryJ: If the new inserted entry did not match () an existing entry in any TLB cache, the TLBIndex.E flag is set and the RC field is cleared to 0. If any single TLB Cache had multiple matching entries, the TLBIndex.E flag is set, the RC is set to 0x4, and the Index field is set to 0; in addition, the multiple matching stale entries are automatically invalidated by the hardware as described in the TLBDeleteEntry command. (The newly inserted entry retains the valid bit that was present in PD0 during the TLBInsertEntry command execution). Present from version 0x04 onwards.

If an uncorrectable ECC error is detected during insert (or any command which reads the JTLB), the E and RC bits are set to 1, 0x7 respectively. The index returned reflects that any cache ways with uncorrectable error behave as if their valid bit was cleared. Thus, if the insert hit two entries in the JTLB but one of those entries had an uncorrectable error detected, then only the other (error free) entry is declared a match; it is replaced with the new entry and its index is returned in TLBIndex, but the E/RC code returned is 1, 0x7.

26.1.14 TLBDeleteEntry

The delete command is used to invalidate specific entries from the OLM TLB caches. The command deletes matching entries in all three TLB caches: the JTLB, μ ITLB, and μ DTLB. The search for entry deletion spans all TLB caches regardless of the presence of the entry in the JTLB. The delete commands consider the TLBPD0 page valid bit in determining the scope of the search for entries to delete. The TLBPD0.V (valid) bit must be set for any entries to be considered a match (and deleted). The matching algorithm of delete operations is identical to that of insert; please see the insert command for details on the entries matched and deleted.

If the delete command does not match (and delete) any existing entry(s) in any TLB cache, the TLBIndex.E flag is set and the RC and Index fields are cleared to 0.

When a delete command finds a single matching entry in JTLB, and the μ TLB did not report multiple match error(s), the TLBIndex register is updated with the index of the matched (and deleted) entry. The TLBIndex E and RC fields are cleared to 0. This is the normal case of entry deletion.

If any single TLB Cache had multiple matching entries, the TLBIndex.E flag is set and the RC is set to 0x4. In this case, all of the multiple matching entries are automatically invalidated by the hardware.

Entries in the μ TLBs are invalidated by clearing their valid bits but the entries are otherwise unchanged. Entries in the RAM based JTLB are invalidated by clearing their valid bits and copying all other fields of TLBPD0 to the matched entries being deleted. The PD1 descriptor of (matched) TLB entries is not modified for any TLB.

If an uncorrectable ECC error is detected during delete (or any command which reads the JTLB), the E and RC bits are set to 1, 0x7 respectively. The index returned reflects that any cache ways with uncorrectable error behave as if their valid bit was cleared. Thus, if the delete hit two entries in the JTLB but one of those entries had an uncorrectable error detected, then only the other (error free) entry is declared a match; it is deleted and its index is returned in TLBIndex, but the E/RC code returned is 1, 0x7.

26.1.15 TLBResetLRU

This command resets the LRU state in μ ITLB and μ DTLB. For JTLB, LRU state and the lock bits are not cleared by the TLBResetLRU command. In this case, software can update the lock bits as necessary using the TLBWrite, TLBInsert, or TLBDelete commands.

LRU state has N entries, each contains the index. After reset the LRU state is {N-1},...,{0}. entry 0 is the Most Recently Used (MRU) and entry N-1 is the Least Recently Used (LRU), where N is the maximum number of allocatable entries.

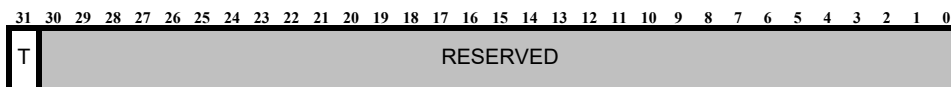
26.1.16 TLB Enable Register, TLBEnable

Address: 0x468

Access: RW

Reset 0x00

Figure 26-11 TLBEnable Register



T, Global TLB Enable

The Global TLB Enable bit is used to enable or disable the OLM. When this bit is set to 0 the OLM is disabled, which means that all virtual addresses are mapped directly to physical addresses. The OLM needs to be enabled (Global TLB Enable bit set to 1) for memory protection and cacheability to work on individual pages.

Note:

- A machine check exception causes the Global TLB enable to be cleared (set to zero).
- Data Uncached region is always active even when the OLM is disabled. This field is set to 0x0 on reset.

Writes to this register are serializing, that is, a pipeline flush occurs after the auxiliary write is committed. For best operating system performance, it is desirable to minimize time spent in TLB miss handlers - hence minimizing pipeline flushes is important. Read operations are not serializing.

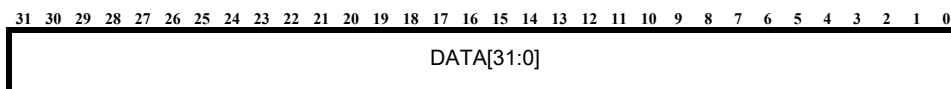
26.1.17 Scratch Data Register, SCRATCH_DATA0

Address: 0x46C

Access: RW

Reset 0x00

Figure 26-12 SCRATCH_DATA0 Register



The `SCRATCH_DATA0` auxiliary register is a generic 32-bit scratch register that can be used in kernel mode only to store any data. The OS can for example use this register to hold the base address of the first level page descriptor table to speed up page table access. The default on reset is 0x0 and writes to the `SCRATCH_DATA0` are non-serializing.

26.2 OLM Build Configuration Register, OLM_BUILD

Address: 0x6F

Access: R

Figure 26-13 OLM_BUILD Register

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
VERSION								RESERVED				PSZ0				RESERVED				JA		JE		R	AS	ITLB			DTLB		

The build configuration register `MMU_BUILD` (0x6F) contains information for the operating system to determine the configuration of the memory management unit. The TLB replacement algorithms are inferred from the version number.

Always access this register in kernel mode. An attempt to access this register from user mode raises a Privilege Violation exception.

Any write to the build configuration register, whether in kernel or user mode, raises an Illegal Instruction exception.

Table 26-5 OLM_BUILD Register Fields

Field		Description
DTLB	[2:0]	Indicates the number of DTLB/ μ DTLB entries 0x1: 4 entries 0x2: 8 entries 0x3: 16 entries All other values are reserved.
ITLB	[5:3]	Indicates the number of ITLB/ μ ITLB entries 0x0 : 2 entries 0x1: 4 entries 0x2: 8 entries All other values are reserved.
AS	[6]	ASID support: Specifies whether ASID is supported or not. This is 0 for this revision.
R	[7]	Reserved, read as zero and ignored on writes.
JE	[9:8]	JTLB entries 0x0: 64 entries 0x1: 128 entries 0x2: 256 entries 0x3: reserved

Table 26-5 OLM_BUILD Register Fields

Field		Description
JA	[11:10]	JTLB ways 0x2: 4-way set associative All other values are reserved.
R	[14:12]	Reserved, read as zero and ignored on writes.
PSZ0	[18:15]	Page Size (PSZ0: normal page size and PSZ1: super page size) 0x3: 4K bytes 0x4: 8K bytes 0x5: 16K bytes All other values are reserved.
R	[23:19]	Reserved, read as zero and ignored on writes.
VERSION	[31:24]	VERSION 0x7: first version of OLM.

27

Program Storage

The non-volatile instruction closely couple memory (NV_ICCM) is an asynchronous non-volatile memory that supports instruction fetch as well as load references, but does not support store or execute access because of the long erase and program times on non-volatile memories. You can program the NV_ICCM program storage using the following auxiliary registers:

27.1 NV_ICCM Register Interface

Use the auxiliary registers in [Table 27-1](#) to program and control the non-volatile ICCM.

Table 27-1 NV_ICCM Registers

Address	Name	Description
0x354	NV_ICCM Control Register, NV_ICCM_CTRL	NV_ICCM control register.
0x355	NV_ICCM Erase Register, NV_ICCM_ERASE	NV_ICCM erase register
0x356	NV_ICCM Programming Register, NV_ICCM_PROG	NV_ICCM programming register
0x357	NV_ICCM Data Register, NV_ICCM_DATA	NV_ICCM data register
0x78	ICCM Configuration Register, ICCM_BUILD	Build Configuration Register

27.1.1 NV_ICCM Control Register, NV_ICCM_CTRL

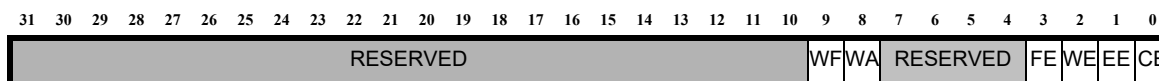
Address: 0x354

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset 0x0

Figure 27-1 NV_ICCM_CTRL Register



Use this register to program the non-volatile ICCM (NV_ICCM).

Table 27-2 NV_ICCM_CTRL Field Descriptions

Field	Bit	Description
CE	[0]	Enable correction exception. When this bit is set to 1, one-bit error corrections raise exceptions.
EE	[1]	Enable ECC exception. When this bit is set to 1, two-bit errors raise exceptions.
WE	[2]	Write enable. Write a 1 to the <code>WE</code> bit to permit write or erase operations on NV_ICCM. If the <code>WE</code> bit is set to 0, and you try to program or erase NV_ICCM using auxiliary registers, the <code>WF</code> bit is set to 1, and the program or erase operation is not executed.
FE	[3]	Program failure exception enable. When this bit is set to 1, the following events raise exceptions: ■ Programming an address that is out of range of the NV_ICCM region. ■ Program failures during the NV_ICCM write and erase operations. When this bit is set to 0, these events do not raise exceptions, but set the <code>NV_ICCM_CTRL.WF</code> bit to 1.
WA	[8]	When the <code>WA</code> bit is set to 1, this bit indicates that a program or erase operation is in progress. Writes to the <code>WA</code> bit are ignored.

Table 27-2 NV_ICCM_CTRL Field Descriptions (Continued)

Field	Bit	Description
WF	[9]	Write or erase failure bit. This bit is sticky. If set to 1, it remains set until you explicitly write a 0. This bit is set to 1 for the following events when <code>NV_ICCM_CTRL.FE==0</code>: ■ Programming an address that is out of range of the NV_ICCM region. ■ Program failures during NV_ICCM write and erase operations. This bit is also set to 1, when <code>NV_ICCM_CTRL.WE==0</code> and you attempt to program or erase NV_ICCM using the programming auxiliary register: <code>NV_ICCM_PROG</code>.

27.1.2 NV_ICCM Erase Register, NV_ICCM_ERASE

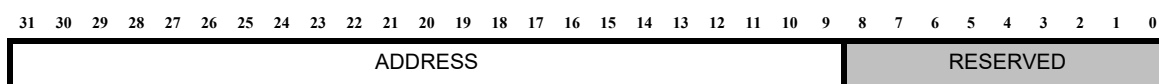
Address: 0x355

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset 0x0

Figure 27-2 NV_ICCM_ERASE Register



When NV_ICCM_CTRL.WA==0, use this register to erase the page (128 words) at the specified address in the NV_ICCM region. If NV_ICCM_CTRL.WA==1, the erase operation is ignored. Hence, ensure that NV_ICCM_CTRL.WA==0 before starting the erase operation. The lower nine bits of this register are ignored on write and read back as zero. If the address written to this register is out of the NV_ICCM region, an illegal instruction exception is raised if NV_ICCM_CTRL.FE==1; if NV_ICCM_CTRL.FE==0, the NV_ICCM_CTRL.WF bit is set to 1. If the address written to this register is in the NV_ICCM region but out of the range of NV_ICCM memory, the specified address is wrapped around.

If the MPU or code protection scheme is enabled, and even if an address protected by the MPU KW attribute is written into this register, the erase operation is executed and an EV_ProtV exception is not raised. You must be cautious when programming this register.

If the ICCM0 or ICCM1 region overlaps with the NV_ICCM region, the erase operation is still executed on the NV_ICCM region.

Table 27-3 NV_ICCM_ERASE Field Descriptions

Field	Bit	Description
ADDRESS	[31:9]	Address of the NV_ICCM region to be erased. If -nv_iccm_ifq==0, 128 words are erased. If -nv_iccm_ifq==1, 512 words are erased.

27.1.3 NV_ICCM Programming Register, NV_ICCM_PROG

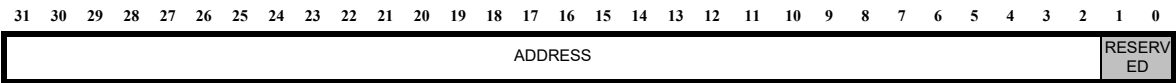
Address: 0x356

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x0

Figure 27-3 NV_ICCM_PROG Register



When NV_ICCM_CTRL.WA==0, use this register to program the specified NV_ICCM memory address. If NV_ICCM_CTRL.WA==1, the program operation is ignored. Hence, ensure that NV_ICCM_CTRL.WA==0 before starting the program operation. If the address written to this register is out of the NV_ICCM region, an illegal instruction exception is raised if NV_ICCM_CTRL.FE==1; if NV_ICCM_CTRL.FE==0, the NV_ICCM_CTRL.WF bit is set to 1. If the address written to this register is in the NV_ICCM region but out of the range of NV_ICCM memory, the specified address is wrapped around.

The data to be programmed into the NV_ICCM is specified by the NV_ICCM_DATA register.

27.1.4 NV_ICCM Data Register, NV_ICCM_DATA

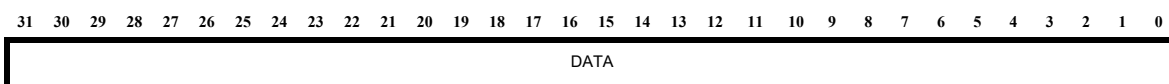
Address: 0x357

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00

Figure 27-4 NV_ICCM_DATA Register



This register specifies the data to be written into NV_ICCM.

27.2 Exceptions

See [Table 6-6](#) on page [268](#).

28

Cryptographic Key Storage

28.1 Cryptographic Key Storage Introduction

The cryptographic key storage is a non-volatile memory that is used by the ARCV2-based processors to store cryptographic keys. The key storage component has only one primary auxiliary interface. The CPU, that does the key calculation and storage uses this auxiliary interface. This design allows reading and writing of the store until the key storage and testing is complete.

When secure MPU is present in the build, and `-mpu_sid_option==true`, the key storage is protected by the secure MPU as follows:

- Verifying the access to the key storage region: if the key storage is defined as a secure region, the MPU verifies the instruction fetch and the data fetch are also secure
- Matching SID. The secure MPU matches the SID specified in this `KEY_STORE_SID` register with the context SID.

When both SID and S violations occur on the key storage memory, S violation takes priority.

28.2 Cryptographic Key Storage Register Interface

Table 28-1 Calibration Storage Registers

Address	Name	Description
0x350	Key Store Control Register, <code>KEY_STORE_CTRL</code>	Key storage control register.
0x351	Key Store Password Register, <code>KEY_STORE_PWD</code>	Key storage password register
0x352	Key Store Address Register, <code>KEY_STORE_ADDR</code>	Key storage address register
0x353	Key Store Data Register, <code>KEY_STORE_DATA</code>	Key storage data register
0x35C	Key Store SID Register, <code>KEY_STORE_SID</code>	Key storage SID register.

Table 28-1 Calibration Storage Registers

Address	Name	Description
0xD8	Key Store Build Configuration Register, KEY_STORE_BUILD	Key storage build configuration register

28.2.1 Key Store Control Register, KEY_STORE_CTRL

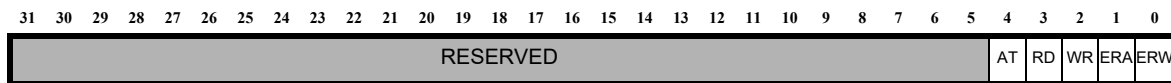
Address: 0x350

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset 0x0

Figure 28-1 KEY_STORE_CTRL Register



Use this register to read, write, and erase the cryptographic key storage non-volatile memory.

When `-sec_modes_option==true`, this register is accessible only in the secure kernel mode. Attempts to access this register from the secure user mode results in a basic privilege violation (0x070000). Attempts to access this register from a normal mode triggers a `EV_PrivilegeV` exception with ECR 0x071020.

Table 28-2 KEY_STORE_CTRL Field Descriptions

Field	Bit	Description
ERW	[0]	Erase 32-bit data. Write a 1 to this bit to erase 32-bit data in the key storage. Before enabling this bit, do the following: 1. Ensure that the <code>KEY_STORE_CTRL.AT</code> bit is set to 0. 2. Write the destination address in the <code>KEY_STORE_ADDR</code> register. 3. To trigger the operation, write the password (default: 0x5A) to the <code>KEY_STORE_PWD</code> register. This bit is write-only. Reads of this bit return 0. Note: If you do not write the password to the <code>KEY_STORE_PWD</code> register, the operation is ignored.
ERA	[1]	Erase an entire array, that is, erase the entire key storage memory. Write a 1 to this bit to erase the entire key storage. Before enabling this bit, do the following: 1. Ensure that the <code>KEY_STORE_CTRL.AT</code> bit is set to 0. 2. Write the destination address in the <code>KEY_STORE_ADDR</code> register. 3. To trigger the operation, write the password (default: 0x5A) to the <code>KEY_STORE_PWD</code> register. This bit is write-only. Reads of this bit return 0. Note: If you do not write the password to the <code>KEY_STORE_PWD</code> register, the operation is ignored.

Table 28-2 KEY_STORE_CTRL Field Descriptions

Field	Bit	Description
WR	[2]	Write enable. Write a 1 to this bit to trigger a write operation to the key storage. Before enabling this bit, do the following: 1. Ensure that the <code>KEY_STORE_CTRL.AT</code> bit is set to 0. 2. Write the destination address in the <code>KEY_STORE_ADDR</code> register. 3. Write the data to the <code>KEY_STORE_DATA</code> register. 4. To trigger the operation, write the password (default: 0x5A) to the <code>KEY_STORE_PWD</code> register. This bit is write-only. Reads of this bit return 0. Note: If you do not write the password to the <code>KEY_STORE_PWD</code> register, the operation is ignored.
RD	[3]	Read enable. Write a 1 to this bit to trigger a read operation from the key storage. Before enabling this bit, do the following: 1. Ensure that the <code>KEY_STORE_CTRL.AT</code> bit is set to 0. 2. Write the source address, from where the data is read, in the <code>KEY_STORE_ADDR</code> register. After the read operation, the data is returned in the <code>KEY_STORE_DATA</code> register. This bit is write-only. Reads of this bit return 0.
AT	[4]	When set to 1, this bit indicates that the key storage is active; write or read operation is in progress. When the key storage is active, no other operations are triggered. This bit is a read-only bit. Writes to this bit are ignored.

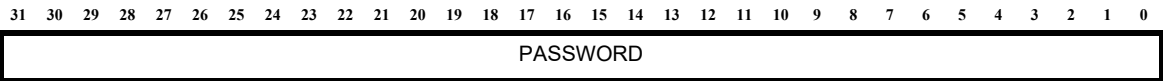
**Note**

- When the `ERW`, `ERA`, `WR`, or `RD` operations are programmed, no further operations are triggered if the `NV_ICCM` is active; you must poll the `AT` bit before triggering any operation.
- You can program only one of the `ERW`, `ERA`, `WR`, or `RD` operations at a time. If you program more than one of these operations, that is, if more than one bit is set in the `KEY_STORE_CTRL[3:0]` field at a time, all the operations are ignored.

28.2.2 Key Store Password Register, KEY_STORE_PWD

- Address: 0x351
- Access:
 - When SecureShield 2+2 mode is not configured: RW
 - When SecureShield 2+2 mode is configured: RW in the secure mode only
- Reset 0x0

Figure 28-2 KEY_STORE_PWD Register



Write the password (default: 0x5A) that grants you privileges to write and erase the key storage. Writing any other value to this register locks the write and erase access to the key storage. You have to write the password to this register for every write and erase action on the key storage.

When `-sec_modes_option==true`, this register is accessible only in the secure kernel mode. Attempts to access this register from the secure user mode results in a basic privilege violation (0x070000). Attempts to access this register from a normal mode triggers a EV_PrivilegeV exception with ECR 0x071020.

Table 28-3 KEY_STORE_PWD Field Descriptions

Field	Bit	Description
PASSWORD	[31:0]	Password that grants you privileges to program and erase the key storage. You can change the password by defining the <code>KEY_STORE_PASSWORD</code> parameter in the RTL. 0x5A is the default value of this local parameter.

28.2.3 Key Store Address Register, KEY_STORE_ADDR

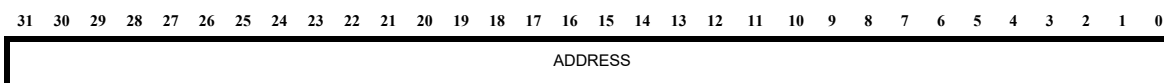
Address: 0x352

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset 0x0

Figure 28-3 KEY_STORE_ADDR Register



This register specifies a key store memory address on which you can perform read, write, or erase operations. If an address is out of range, the address is wrapped around. The operation that the processor performs on this address is specified by the KEY_STORE_CTRL register.

When `-sec_modes_option==true`, this register is accessible only in the secure kernel mode. Attempts to access this register from the secure user mode results in a basic privilege violation (0x070000). Attempts to access this register from a normal mode triggers a EV_PrivilegeV exception with ECR 0x071020.

28.2.4 Key Store Data Register, KEY_STORE_DATA

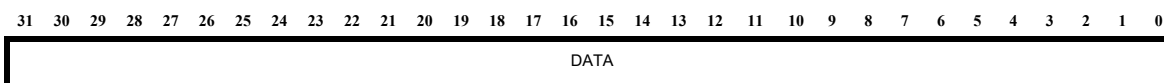
Address: 0x353

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00

Figure 28-4 KEY_STORE_DATA Register



For a key store memory read operation, this register returns the data read from the memory. For a key store write operation, this register stores the data that is written to the memory.

The read or write operation is determined by the value of the KEY_STORE_CTRL register.

When `-sec_modes_option==true`, this register is accessible only in the secure kernel mode. Attempts to access this register from the secure user mode results in a basic privilege violation (0x070000). Attempts to access this register from a normal mode triggers a EV_PrivilegeV exception with ECR 0x071020.

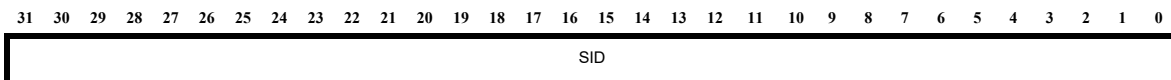
28.2.5 Key Store SID Register, KEY_STORE_SID

Address: 0x35C

Access: RW in the secure mode

Reset: 0x00

Figure 28-5 KEY_STORE_SID Register



Use this register to program the secure ID context or contexts in which the key storage can be used in your application. This register is present in the build only when the `-mpu_sid_option==true`.

When secure MPU is present in the build, and `-mpu_sid_option==true`, the key storage is protected by the secure MPU as follows:

- Verifying the access to the key storage region: if the key storage is defined as a secure region, the MPU verifies the instruction fetch and the data fetch are also secure
- Matching SID. The secure MPU matches the SID specified in this KEY_STORE_SID register with the context SID.

When both SID and S violation occur on the key storage memory, S violation take priority.

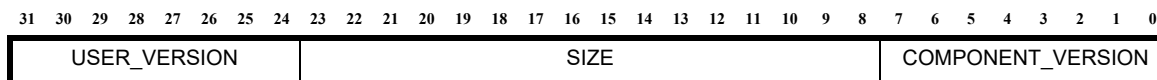
This register is accessible only in the secure kernel mode. Attempts to access this register from the secure user mode results in a basic privilege violation (0x070000). Attempts to access this register from a normal mode triggers a EV_PrivilegeV exception with ECR 0x071020.

28.2.6 Key Store Build Configuration Register, KEY_STORE_BUILD

Address: 0xD8

Access: R

Figure 28-6 KEY_STORE_BUILD Register



This register contains the cryptographic key storage configuration.

Table 28-4 KEY_STORE_BUILD

Field	Bit	Description
COMPONENT_VERSION	[7:0]	Version of the cryptographic key-storage component 0x01: initial version 0x02: version when -sec_modes_option==true. This enables protection of the key storage by the secure MPU. A register KEY_STORE_SID is added when -mpu_sid_option==true.
SIZE	[23:8]	Size in 32-bit words
USER_VERSION	[31:24]	Version of the cryptographic key storage component, configurable in ARCHitect

29

Calibration Parameter Storage

The calibration parameter storage is a non-volatile storage in the ARCV2-processors. You can program this memory using the following auxiliary registers.

Table 29-1 Calibration Storage Registers

Address	Name	Description
0x358	CAL_STORE_CTRL	Calibration storage control register.
0x359	CAL_STORE_ADDR	Calibration storage address register
0x35A	CAL_STORE_DATA	Calibration storage data register
0xD9	CAL_STORE_BUILD	Build configuration register for calibration storage

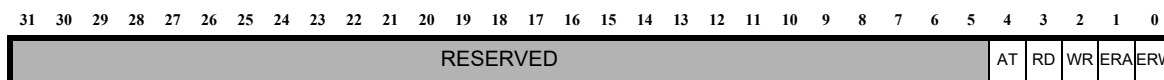
29.0.1 Calibration Parameter Store Control Register, CAL_STORE_CTRL

Address: 0x358

Access: RW

Reset 0x0

Figure 29-1 CAL_STORE_CTRL Register



Use this register to read, write, and erase the calibration parameter storage non-volatile memory.

Table 29-2 CAL_STORE_CTRL Field Descriptions

Field	Bit	Description
ERW	[0]	Erase 32-bit data. Write a 1 to this bit to erase 32-bit data in the calibration parameter storage. Before enabling this bit, do the following: 1. Ensure that the <code>CAL_STORE_CTRL.AT</code> bit is set to 0. 2. Write the destination address in the <code>CAL_STORE_ADDR</code> register. This bit is write-only. Reads to this bit return 0.
ERA	[1]	Write a 1 to this bit to erase the entire calibration parameter storage. Before enabling this bit, do the following: 1. Ensure that the <code>CAL_STORE_CTRL.AT</code> bit is set to 0. 2. Write the destination address in the <code>CAL_STORE_ADDR</code> register. This bit is write-only. Reads of this bit return 0.
WR	[2]	Write enable. Write a 1 to this bit to trigger a write operation to the calibration parameter storage. Before enabling this bit, do the following: 1. Ensure that the <code>CAL_STORE_CTRL.AT</code> bit is set to 0. 2. Write the destination address in the <code>CAL_STORE_ADDR</code> register. 3. Write the data to the <code>CAL_STORE_DATA</code> register. This bit is write-only. Reads of this bit return 0.

Table 29-2 CAL_STORE_CTRL Field Descriptions

Field	Bit	Description
RD	[3]	Read enable. Write a 1 to this bit to trigger a read operation from the calibration parameter storage. Before enabling this bit, do the following: 1. Ensure that the <code>CAL_STORE_CTRL.AT</code> bit is set to 0. 2. Write the source address, from where the data is read, in the <code>CAL_STORE_ADDR</code> register. After the read operation, the data is returned in the <code>CAL_STORE_DATA</code> register. This bit is write-only. Reads of this bit return 0.
AT	[4]	When this bit is set to 1, this bit indicates that a program or erase operation is in progress. When the calibration parameter storage is active, no other operations are triggered. This bit is a read-only bit. Writes to this bit are ignored.

**Note**

- When the `ERW`, `ERA`, `WR`, or `RD` operations are programmed, no further operations are triggered if the `NV_ICCM` is active; you must poll the `AT` bit before triggering any operation.
- You can program only one of the `ERW`, `ERA`, `WR`, or `RD` operations at a time. If you program more than one of these operations, that is, if more than one bit is set in the `KEY_STORE_CTRL[3:0]` field at a time, all the operations are ignored.

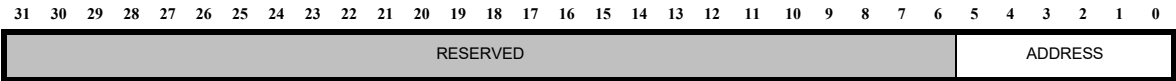
29.0.2 Calibration Parameter Store Address Register, CAL_STORE_ADDR

Address: 0x359

Access: RW

Reset: 0x0

Figure 29-2 CAL_STORE_ADDR Register



This register specifies a calibration parameter store memory address on which you can perform the read, write, or erase operations. If an address is out of range, the address is wrapped around. The operation that the processor performs on this address is specified by the CAL_STORE_CTRL register.

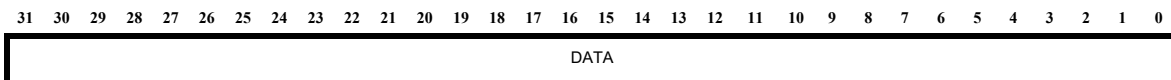
29.0.3 Calibration Parameter Store Data Register, CAL_STORE_DATA

Address: 0x35A

Access: RW

Reset: 0x00

Figure 29-3 CAL_STORE_DATA Register



For a calibration parameter store memory read operation, this register returns the data read from the memory. For a write operation, this register stores the data that will be written to the memory.

The read or write operation is determined by the value of the CAL_STORE_CTRL register.

29.1 Build Configuration Registers

The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

A set of baseline Build Configuration Registers are present in all ARCV2 processors. In addition, each optional ISA extension typically includes a Build Configuration Register to signify its presence and its specific configuration in each given build.

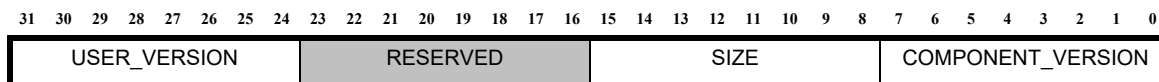
Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCV2-based system.

29.1.1 Calibration Parameter Store Build Configuration Register, CAL_STORE_BUILD

Address: 0xD9

Access: R

Figure 29-4 CAL_STORE_BUILD Register



This register contains the calibration parameter storage configuration.

Table 29-3 CAL_STORE_BUILD

Field	Bit	Description
COMPONENT_VERSION	[7:0]	Calibration parameter storage unit version 0x01: Current version
SIZE	[15:8]	Size in 32-bit words
USER_VERSION	[31:24]	Version of the calibration parameter storage component, configurable in ARCHitect

30

Processor Timers

The processor timers are two independent 32-bit timers and a 64-bit real-time counter (RTC). Timer 0 and timer 1 are identical in operation. The only difference is that these timers are connected to different interrupts. The Timers cannot be included in a configuration without interrupts. Each timer is optional and when present, it is connected to a fixed interrupt; interrupt 16 for timer 0 and interrupt 17 for timer 1.

The processor timers include:

- two independent 32-bit timers and a 64-bit real-time counter (RTC) in the normal mode. Timer 0 and timer 1 are identical in operation. The only difference is that these timers are connected to different interrupts. The Timers cannot be included in a configuration without interrupts. Each timer is optional and when present, it is connected to a fixed interrupt; interrupt 16 for timer 0 and interrupt 17 for timer 1.
- when `-sec_modes_option==true`, two secure timers are included: ST0 (included if `-sec_timer_0==true`) and ST1 (included if `-sec_timer_1==true`). ST0 is connected to interrupt 20 and ST1 is connected to ST21.

30.1 Timers

The processor timers are connected to a system clock signal that operates even when the ARCV2-based processor is in some (but not all) of the processor sleep states. The watchdog timer, if configured, can be connected to the same system clock as the other timers, or an external clock signal. In the sleep states where the timers are enabled, the timers can be used to generate interrupt signals that wake the processor from the SLEEP state. For more information about internal clocks to the timers during the sleep state, see [Table 33-3](#) on page [1035](#).



Note

When the processor is in the HALT state, timers do not count up or down, but the registers can be accessed by the debugger. This means that the timer interrupts (including a watchdog interrupt, if the watchdog is configured to be internally clocked), are not generated when the processor is halted.

The processor timers automatically reset and restart their operation after reaching the limit value. The processor timers can be programmed to count only the clock cycles when the processor is not halted. The processor timers can also be programmed to generate an interrupt or to generate a system [Reset](#) upon reaching the limit value.

The 64-bit RTC does not generate any interrupts. This timer is used to count the clock cycles atomically.

30.1.0.1 Programming

To program a timer *n*, use the following sequence:

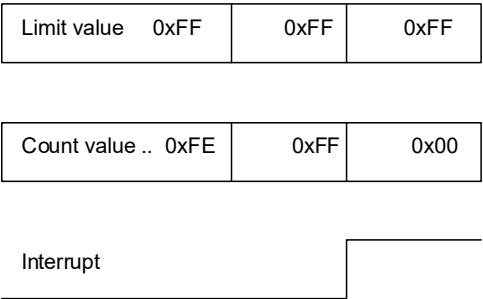
- Write 0 to the CONTROL_{*n*} register to disable interrupts.
- Write the limit value to the timer LIMIT_{*n*} register.
- Set up the control flags according to the desired mode of operation by updating the timer CONTROL_{*n*} register.
- Write the count value to the timer count_{*n*} register.

Timer *n* starts counting upwards from the COUNT_{*n*} value to the LIMIT_{*n*} value, and an interrupt is generated if interrupts are enabled. Timer *n* then automatically restarts to count upwards from 0 to the limit value.

 **Note**

When you are using the secure timers and normal timers, ensure that the minimum number of interrupt priority levels is at least two. And, the secure timers interrupts have higher priority compared to the normal timers.

Figure 30-1 Interrupt Generated after Timer Reaches Limit Value



The software must clear the timer interrupts by clearing CONTROL_{*n*}.IP bit. After an interrupt is generated, write to the CONTROL_{*n*} register to clear the timer count. This action must be performed during the interrupt service routine.

30.2 Auxiliary Timer Registers

The ARCV2 architecture provides one or two optional cycle counters, each of which defines three additional auxiliary registers as listed in Table 30-1 and Table 30-2.

Table 30-1 Auxiliary Registers for Hardware Counter 0 (has_timer_0 == true)

Address	Auxiliary Register Name	Description
0x21	Timer 0 Count Register, COUNT0	Processor timer 0 count value
0x22	Timer 0 Control Register, CONTROL0	Processor timer 0 control value

Table 30-1 Auxiliary Registers for Hardware Counter 0 (has_timer_0 == true) (Continued)

Address	Auxiliary Register Name	Description
0x23	Timer 0 Limit Register, LIMIT0	Processor timer 0 limit value

Table 30-2 Auxiliary Registers for Hardware Counter 1 (has_timer_1 == true)

Address	Auxiliary Register Name	Description
0x100	Timer 1 Count Register, COUNT1	Processor timer 1 count value
0x101	Timer 1 Control Register, CONTROL1	Processor timer 1 control value
0x102	Timer 1 Limit Register, LIMIT1	Processor timer 1 limit value

Table 30-3 Auxiliary Register for Real-time counter

Address	Auxiliary Register Name	Description
0x103	RTC Control Register, AUX_RTC_CTRL	RTC control register
0x104	RTC Count Low Register, AUX_RTC_LOW	RTC count low register
0x105	RTC Count High Register, AUX_RTC_HIGH	RTC count high register

Table 30-4 Auxiliary Registers for Hardware Counter 0 (-sec_timer_0==true)

Address	Auxiliary Register Name	Description
0x106	Secure Timer 0 Count Register, AUX_ST0_COUNT	Processor timer 0 count value
0x107	Secure Timer 0 Control Register, AUX_ST0_CTRL	Processor timer 0 control value
0x108	Secure Timer 0 Limit Register, AUX_ST0_LIMIT	Processor timer 0 limit value

Table 30-5 Auxiliary Registers for Hardware Counter 0 (-sec_timer_1==true)

Address	Auxiliary Register Name	Description
0x109	Secure Timer 1 Count Register, AUX_ST1_COUNT	Processor timer 0 count value
0x10A	Secure Timer 1 Control Register, AUX_ST1_CTRL	Processor timer 0 control value
0x10B	Secure Timer 1 Limit Register, AUX_ST1_LIMIT	Processor timer 0 limit value

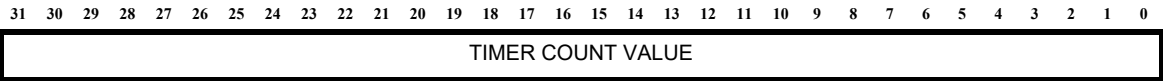
30.2.1 Timer 0 Count Register, COUNT0

Address: 0x21

Access: RW

Reset: 0x0000_0000

Figure 30-2 Timer 0 Count Value Register



Writing to this register sets the initial count value for the timer, and restarts the timer. Subsequently, the register can be read to reflect the timer 0 count progress.

The COUNT0 register can be updated when the timer is running in which case the internal count register is updated with the new count value and the timer starts counting up from the updated value.

This register contains 0x00000000 on reset.

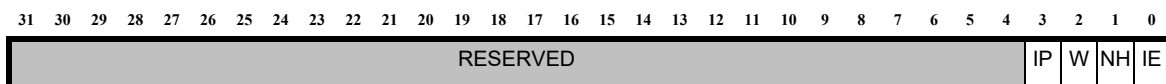
30.2.2 Timer 0 Control Register, CONTROL0

Address: 0x22

Access: RW

Reset: 0x0000_0000

Figure 30-3 Timer 0 Control Register



The timer control register (CONTROL0) is used to update the control modes of the timer.

Writing to CONTROL0 de-asserts the timer interrupt, but does not stop the timer from counting. The timer continues counting and independently starts the next iteration of counting, setting COUNT0 to 0, when LIMIT0 equals COUNT0.

If both the IE and W bits are set, only the watchdog reset is activated because the ARCV2-based processor has been reset and the interrupt is lost. Regardless of the IE and W bits, the timer is automatically reset and the timer restarts its operation after reaching the limit value.

All of the control flags must be programmed in one write access to the CONTROL0 register.

Table 30-6 TIMER 0 Field Descriptions

Field	Bit	Reset	Description
IE	[0]	0	The Interrupt Enable flag (IE) enables the generation of an interrupt after the timer has reached its limit condition. If this bit is not set, no interrupt is generated. When the processor is Reset , the IE flag is set to 0.
NH	[1]	0	The Not Halted mode flag (NH) causes cycles to be counted only when the processor is running (that is when the processor is not halted). When set to 0, the timer counts every clock cycle. When set to 1, the timer counts only when the processor is running. If the NH flag is set to 1, counting is suspended during host debugger interactions with the processor. However, if the NH flag is set to 0, counters are free-running. When the processor is Reset , the NH flag is set to 0.
W	[2]	0	The Watchdog mode flag (W) enables the generation of a system watchdog reset signal after the timer has reached its limit condition. If this bit is not set, no watchdog reset signal is generated. The watchdog reset signal is activated two cycles after the limit condition is reached. The watchdog reset signal can be used to cause a system or processor Reset with appropriate external logic.

Table 30-6 TIMER 0 Field Descriptions (Continued)

Field	Bit	Reset	Description
IP	[3]	0	The IP bit is set when the COUNTn register reaches the LIMITn value, and remains set until cleared by the timer interrupt service routine. The IP bit can be cleared by re-writing the desired values of W, NH and IE into the CONTROL0 register, thereby writing a 0 into the IP position. An interrupt service routine typically clears the IP bit by reading the CONTROL0 register, masking out the IP bit, and writing the resulting value back to the CONTROL0 register.
RESERVED	[31:5]	0	Reserved. Read-as-zero; ignored on write.

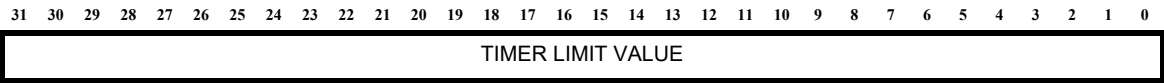
30.2.3 Timer 0 Limit Register, LIMIT0

Address: 0x23

Access: RW

Reset: 0x00FFFFFF

Figure 30-4 Timer 0 Limit Value Register

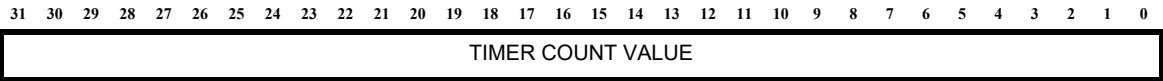


You must write the limit value into this register. The limit value is the value after which an interrupt or a reset must be generated. For backward compatibility to previous processor variants, the timer limit register is set to 0x00FFFFFF when the processor is [Reset](#).

30.2.4 Timer 1 Count Register, COUNT1

Address: 0x100
Access: RW
Reset: 0x0000_0000

Figure 30-5 Timer 1 Count Value Register



See [Timer 0 Count Register, COUNT0](#) on [page 996](#) for field information.

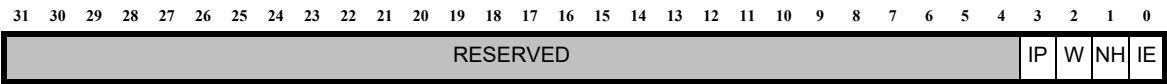
30.2.5 Timer 1 Control Register, CONTROL1

Address: 0x101

Access: RW

Reset: 0x0000_0000

Figure 30-6 Timer 0 Control Register

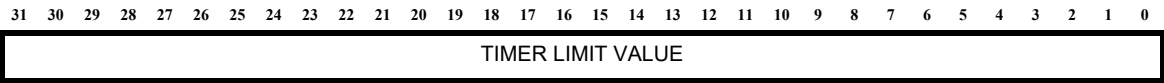


See [Timer 0 Control Register, CONTROL0](#) on [page 997](#) for field information.

30.2.6 Timer 1 Limit Register, LIMIT1

Address: 0x102
Access: RW
Reset: 0x00FFFFFF

Figure 30-7 Timer 1 Limit Value Register



See [Timer 0 Limit Register, LIMIT0](#) on [page 999](#) for field information.

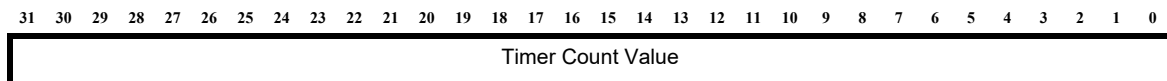
30.2.7 Secure Timer 0 Count Register, AUX_ST0_COUNT

Address: 0x106

Access: RW in the secure mode only

Reset: 0x0000_0000

Figure 30-8 Secure Timer 0 Count Value Register



Writing to this register sets the initial count value for the secure timer 0, and restarts the secure timer 0. Subsequently, the register can be read to reflect the timer 0 count progress.

The secure timer 0 count register can be updated when the secure timer is running in which case the internal count register is updated with the new count value and the secure timer starts counting up from the updated value.

This register contains 0x00000000 on reset.

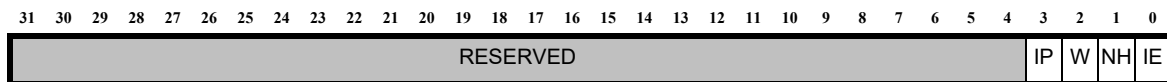
30.2.8 Secure Timer 0 Control Register, AUX_ST0_CTRL

Address: 0x107

Access: RW in the secure mode only

Reset: 0x0000_0000

Figure 30-9 Secure Timer 0 Control Register



The secure timer control register (CONTROL) is used to update the control modes of the timer.

Writing to this register de-asserts the secure timer 0 interrupt, but does not stop the timer from counting. The timer continues counting and independently starts the next iteration of counting, setting secure timer 0 count to 0, when secure timer 0 limit equals the secure timer 0 count.

The Interrupt Enable flag (IE) enables the generation of an interrupt after the secure timer has reached its limit condition. If this bit is not set, no interrupt is generated. When the processor is [Reset](#), the IE flag is set to 0.

The Not Halted mode flag (NH) causes cycles to be counted only when the processor is running (that is when the processor is not halted). When set to 0, the secure timer counts every clock cycle. When set to 1, the timer counts only when the processor is running. If the NH flag is set to 1, counting is suspended during host debugger interactions with the processor. However, if the NH flag is set to 0, counters are free-running. When the processor is [Reset](#), the NH flag is set to 0.

All of the control flags must be programmed in one write access to the secure timer 0 control register.

The IP bit is set when the secure timer count register reaches the secure timer limit value, and remains set until cleared by the secure timer interrupt service routine. The IP bit can be cleared by re-writing the desired values of NH and IE into the secure timer control register, thereby writing a 0 into the IP position. An interrupt service routine typically clears the IP bit by reading the secure timer control register, masking out the IP bit, and writing the resulting value back to the control register.

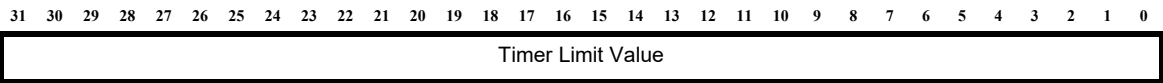
30.2.9 Secure Timer 0 Limit Register, AUX_ST0_LIMIT

Address: 0x108

Access: RW in the secure mode only

Reset: 0x00FFFFFF

Figure 30-10 Secure Timer 0 Limit Value Register



You must write the limit value into this register. The limit value is the value after which an interrupt or a reset must be generated. The timer limit register is set to 0x00FFFFFF when the processor is [Reset](#).

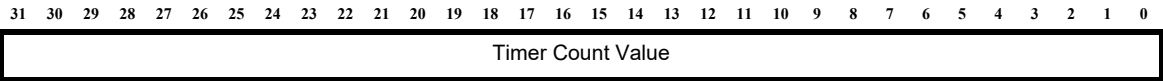
30.2.10 Secure Timer 1 Count Register, AUX_ST1_COUNT

Address: 0x109

Access: RW in the secure mode only

Reset: 0x0000_0000

Figure 30-11 Secure Timer 1 Count Value Register



See “Secure Timer 0 Count Register, AUX_ST0_COUNT” on page 1003 for field information.

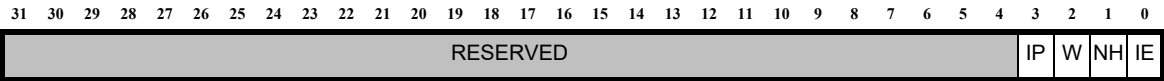
30.2.11 Secure Timer 1 Control Register, AUX_ST1_CTRL

Address: 0x10A

Access: RW in the secure mode only

Reset: 0x0000_0000

Figure 30-12 Secure Timer 1 Control Register



See “Secure Timer 0 Control Register, AUX_ST0_CTRL” on page 1004 for field information.

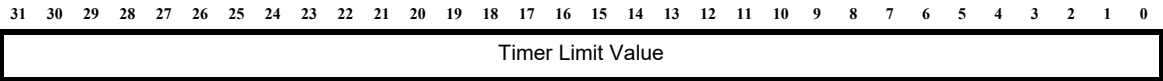
30.2.12 Secure Timer 1 Limit Register, AUX_ST1_LIMIT

Address: 0x10B

Access: RW in the secure mode only

Reset: 0x00FFFFFF

Figure 30-13 Secure Timer 1 Limit Value Register



See “Secure Timer 0 Limit Register, AUX_ST0_LIMIT” on page 1005 for field information.

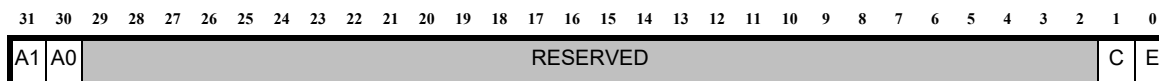
30.2.13 RTC Control Register, AUX_RTC_CTRL

Address: 0x103

Access: rW

Reset: 0x0000_0000

Figure 30-14 AUX_RTC_CTRL Register

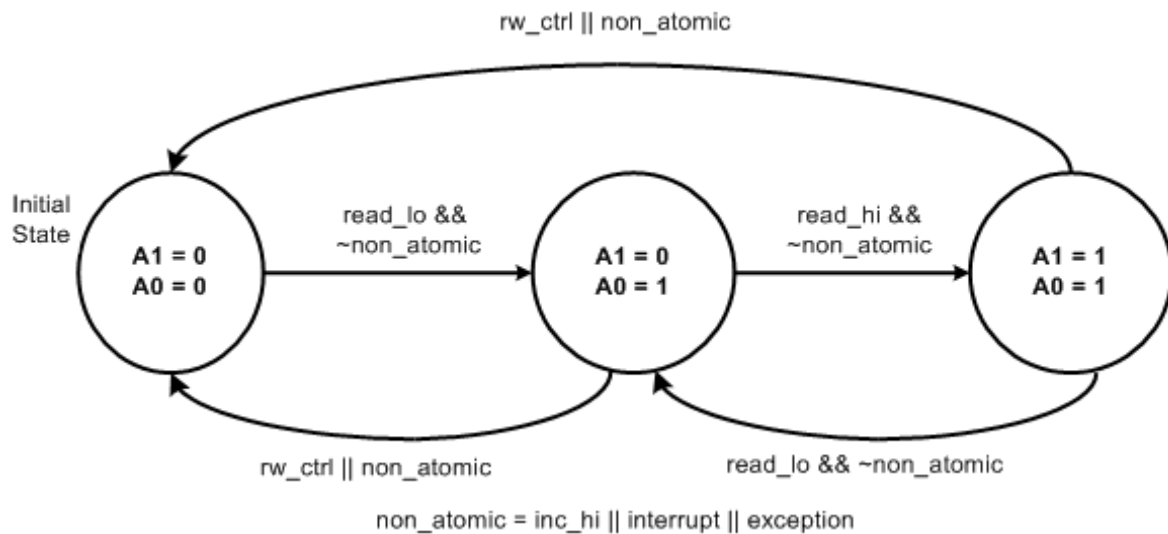


The 64-bit RTC is a free-running counter. This counter does not have any associated interrupts. The RTC is controlled using the AUX_RTC_CTRL register. The following are the field descriptions:

Table 30-7 RTC Control Register

Fields	Bit	Reset	Description
E	[0]	0	Enable A value of 0 means disabled; 1 means enable counting.
C	[1]	0	A value of 1 clears the AUX_RTC_LOW and AUX_RTC_HIGH registers.
RESERVED	[29:2]	0	Reserved. Read-as-zero; ignored on write.
A1, A0	[31:30]	0	These bits track the atomicity of reads from the RTC Count High Register, AUX_RTC_HIGH and RTC Count Low Register, AUX_RTC_LOW registers. For the description of the state machine, see Figure 30-15 on page 1010.

The following state machine describes the values of A1 and A0 based on the reading of the RTC count high and RTC count low registers.

Figure 30-15 State Machine of A1 and A0 bits in RTC Control Register**Note**

For conditions not explicitly shown in the diagram, there is no transition to another state (stay in current state).

30.2.13.1 Reading RTC Atomically

The following example code explains how to read the RTC atomically:

```

RTC_read: LR R0, [AUX_RTC_LOW]
          LR R1, [AUX_RTC_HIGH]
          LR R2, [AUX_RTC_CTRL]
          BBIT0.NT R2, 31, RTC_read

```

- If an interrupt occurs between the first two LR instructions, R2 does not have bit 31 set to 1, and the BRLT loops back to enforce a re-read.
- If an exception occurs when executing any of the LR instructions, a loop back is enforced to re-read.
- If AUX_RTC_HIGH is incremented after reading AUX_RTC_LOW, again a re-read is enforced.
- Due to the infrequency of interrupts and increments to AUX_RTC_HIGH, this loop rarely requires more than 1 iteration.
- As the branch is rarely taken, a static prediction of .NT must be given. This implementation improves the performance on implementations with deeper pipelines.

**Note**

Placing the reads inside a critical region ensures atomicity, but does not detect when the AUX_RTC_LOW register wraps around between the two reads. Therefore, always use the above scheme.

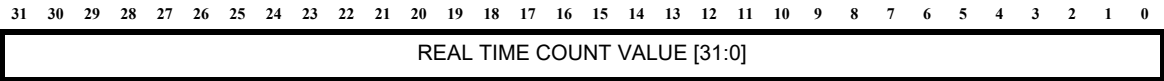
30.2.14 **RTC Count Low Register, AUX_RTC_LOW**

Address: 0x104

Access: r

Reset: 0x0000_0000

Figure 30-16 RTC Count Low Register



The RTC count low register is a read register in user mode. Reading this register returns the LSB 32-bits of the free-running RTC.

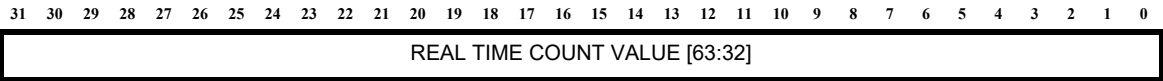
30.2.15 RTC Count High Register, AUX_RTC_HIGH

Address: 0x105

Access: r

Reset: 0x0000_0000

Figure 30-17 AUX_RTC_HIGH Register



The RTC count high register is a read register in user mode. Reading this register returns the MSB 32-bits of the free-running RTC.

30.2.16 Timers Configuration Register, TIMER_BUILD

Address: 0x75

Access: R

The TIMER_BUILD configuration register (TIMER_BUILD) indicates the presence of the processor timers auxiliary registers.

Figure 30-18 TIMER_BUILD Register

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
SP1				SP0				P1				P0				RESERVED		ST1	ST0	RTC	T1	T0	VERSION								

Table 30-8 TIMER_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Current version. All non-enumerated values are reserved. 0x1: Version 1 0x2: ARC 700 Processor Timers 0x3: ARC 600 R3 Processor Timers, with interrupt-pending bits 0x4: ARCV2 Processor Timers 0x05: ARCV2 with secure Timers
T0	[8]	Timer 0 Present 0x0: No timer 0 0x1: Timer 0 present
T1	[9]	Timer 1 Present 0x0: No timer 1 0x1: Timer 1 present
RTC	[10]	64-bit RTC Configuration 0x0: 64-bit RTC is disabled. 0x1: 64-bit RTC is enabled.
ST0	[11]	Secure timer 0 0x0: No secure timer 0 0x1: Secure timer 0 is present
ST1	[12]	Secure timer 1 0x0: No secure timer 1 0x1: Secure timer 1 is present

Table 30-8 **TIMER_BUILD Field Descriptions (Continued)**

Field	Bit	Description
RESERVED	[15:13]	Reserved. Read-as-zero; ignored on write.
P0	[19:16]	Indicates the interrupt priority level of Timer 0 Note: If Timer 0 is not included, this field is always set to 0.
P1	[23:20]	Indicates the interrupt priority level of Timer 1 Note: If Timer 1 is not included, this field is always set to 0.
SP0	[27:24]	Indicates the interrupt priority level of Secure Timer 0 Default: priority level 1
SP1	[31:28]	Indicates the interrupt priority level of Secure Timer 1 Default: priority level 0

31

Enhanced Watchdog Timer

The watchdog timer is used to detect time-based errors as opposed to non-time-based errors (a random bit flip in memory). The timer is used to detect these errors accurately providing a way to initiate a system reset or interrupt in response, thereby invoking some sort of recovery process.

A common time-based error is a system deadlock where some unexpected random event has caused the system to hang. This error could be hardware inflicted, for example, a bus interface locks up, or software inflicted where the processor gets stuck in a non-exiting loop. The ability to recover from such situations is critical in safety systems.

To verify the integrity of the watchdog timer an optional running parity check is available on all defined registers.

The watchdog timer supports the following modes of operation:

- Countdown timer mode
 - In this mode, the timer counts down from a fixed start value (defined in `EWDT_PERIODx`). The software must ensure the timer never reaches the value 0 (zero) by resetting it back to the start value. If the timer reaches the value 0 (zero), a time out occurs and an error is raised.

31.1 Watchdog Timer Index

When more than one watchdog timer is configured in the core, the watchdog registers for each watchdog timer share the same auxiliary space. Therefore, the watchdog registers are banked for each watchdog timer. To program the registers for a specific watchdog timer, you must first select the watchdog register bank of that watchdog timer. To select a watchdog timer bank, you must write the bank index to the `EWDT_INDEX` register.

Writes to a watchdog timer register is performed as follows:

1. Write the index of the required watchdog timer in the `EWDT_INDEX` register.
2. Write to the desired timer auxiliary register.

Reads from a watchdog timer register is performed as follows:

1. Write the index of the required watchdog timer into in the `EWDT_INDEX` register.
2. Read the desired timer auxiliary register.

31.2 Registers

The watchdog timer includes the following registers:



Note

When the watchdog timer is configured with a separate clock, reading or writing to these banked registers may take multiple core clock cycles to complete. The number of cycles is dependent on the clock frequency used to drive the watchdog timer.

Table 31-1 AUX Direct + Index

Register Name	Index	Address
EWDT_PASSWD		0xE00
EWDT_ESR		0xE01
EWDT_INDEX		0xE02
EWDT_CTRL0	INDEX - 0x0	0xE03
EWDT_PERIOD0		0xE04
EWDT_COUNT0		0xE05

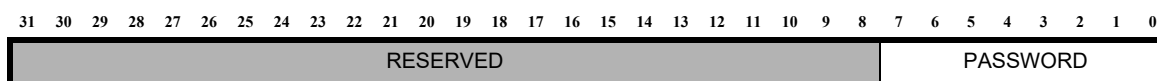
31.2.1 Watchdog Password Register, EWDT_PASSWD

Address: 0xE00

Access: W

Default: N/A

Figure 31-1 EWDT_PASSWD Register



The watchdog password register is used to unlock access to the following watchdog timer registers: WDT_INDEX, EWDT_CTRL, EWDT_PERIOD, EWDT_LTHRESH, EWDT_UTHRESH, and EWDT_ESR. Each watchdog timer register is assigned an unique password which must be used to unlock it for write access. The passwords are defined in [Table 31-2](#).



Note

You cannot use the AEX instruction to access this register.

Table 31-2 EWDT_PASSWORD Field Description

Field	Bits	Description
Password	[7:0]	Password access to the watchdog time auxiliary registers ■ 0xAA: Password to unlock EWDT_CTRL for write access. ■ 0x53: Password to unlock EWDT_ESR for write access. ■ 0x54: Password to unlock EWDT_INDEX for write access. ■ 0x55: Password to unlock EWDT_PERIOD for write access. ■ 0x56: Password to unlock EWDT_LTHRESH for write access. ■ 0x57: Password to unlock EWDT_UTHRESH for write access. ■ 0x5A: In Countdown timer mode (EWDT_CTRL[5:4] = 0x0), this password automatically loads the value in EWDT_PERIOD to the timer's counter.
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

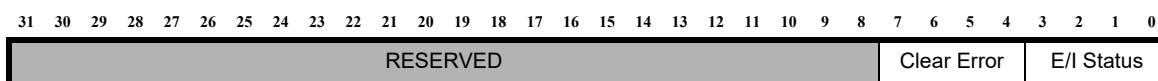
31.2.2 Watchdog Time Error Status Register, EWDT_ESR

Address: 0xE01

Access: RW

Default: 0x0

Figure 31-2 EWDT_ESR Register



The watchdog timer error status register is used to query the error/interrupt status. When a time out error occurs, the respective error bits are set to 1. Writing a 1 to bits [7:4] clears the respective error bits once they are set.

- You must unlock this register before writing to it. You can unlock the register by writing the correct pass code to EWDT_PASSWD. Otherwise, an illegal instruction exception is raised.
- After a write, this register is locked.



Note

- The register does not exist if only one watchdog timer is configured. In a single watchdog timer configuration, the error status is cleared using bit 3 of EWDT_CTRLx.
- You cannot use the AEX instruction to access this register.

Table 31-3 EWDT_ESR Bit Definition

Field	Bits	Reset	Description
E/I Status	[3:0]	0	These bits report the timeout status for all timers. This field is read only. Writes to this field are ignored.
Clear Error	[7:4]	0	These bits are used to clear the error status [3:0] for the corresponding counter.
RESERVED	[31:8]	0	Reserved. Read-as-zero; ignored on write.

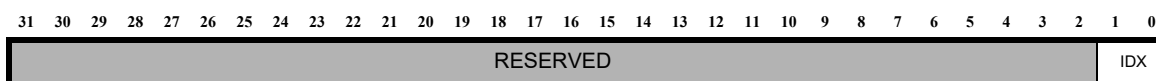
31.2.3 Watchdog Timer Index Register, EWDT_INDEX

Address: 0xE02

Access: RW

Default: 0x0

Figure 31-3 EWDT_INDEX Register



The watchdog timer index register is used to select a timer's registers for read/write access. The index value represents the timer number; for example writing the value 3 allows access to watchdog timer 3 registers. It is not necessary to write the register again until you want to access the registers of a different timer.



Note

You cannot use the AEX instruction to access this register.

Table 31-4 EWDT_INDEX Field Definition

Field	Bits	Reset	Description
Index (IDX)	[1:0]	0	These bits are an index and identify which watchdog timer registers to read/write.
RESERVED	[31:2]	0	Reserved. Read-as-zero; ignored on write.

- You must unlock this register before writing to it. You can unlock the register by writing the correct pass code to the EWDT_PASSWD register. Otherwise, an illegal instruction exception is raised.
- After a write, this register is locked.



Note

Only the registers associated with the timer selected by EWDT_INDEX can be unlocked using the EWDT_PASSWD for write access. When writing a value to this register to access a different watchdog timer any unlocked registers from the currently selected timer are locked again. If the index value is set to a timer that does not exist, then any access to the non-existent EWDT registers result in an illegal instruction exception. This register does not exist if only one watchdog timer is configured.

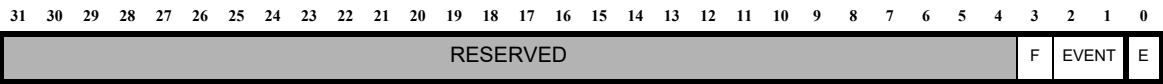
31.2.4 Watchdog Control Register, EWDT_CTRLx

Address: 0xE03 - (Indexed for timers 0-3)

Access: RW

Default: 0x00

Figure 31-4 EWDT_CTRLx Register



The watchdog control register is used to program a watchdog timer.

- You must unlock this register before writing to it. You can unlock the register by writing the correct pass code to the EWDT_PASSWD register. Otherwise, an illegal instruction exception is raised.
- After a write, this register is locked.

 **Note** You cannot use the AEX instruction to access this register.

Table 31-5 EWDT_CTRLx Bit Definition

Field	Bits	Reset	Description
E	[0]	0	Enable watchdog timer (the internal counter begins decrementing in the countdown mode and the window timer mode). ■ 0x0: Counting is disabled ■ 0x1: Counting is enabled

Field	Bits	Reset	Description
EVENT	[2:1]	0	Specifies the action of the watchdog timer when a time out error occurs. 0x0: Assert (active high) the external signal <code>wdt_ext_timeout_r</code>. In this mode, the external input signal <code>wdt_ext_timeout_ack_r</code> is used to acknowledge the external time out. The resulting handshake de-asserts <code>wdt_ext_timeout_r</code>. Only one timer is allowed to be configured in this mode. A timer in this mode cannot exist with a timer set in mode 0x3. 0x1: Assert a level interrupt. Write to the corresponding flag bit in <code>EWDT_CTRL</code> to clear the interrupt. Read the status register <code>EWDT_ESR</code> to determine which watchdog has timed out and generated the interrupt. 0x2: Assert (active high) external reset enable signal <code>wdt_reset</code>. Write to the corresponding flag bit in <code>EWDT_CTRL</code> to clear the reset error. Read the status register <code>EWDT_ESR</code> to determine which watchdog has generated the reset timeout. 0x3: Assert (active high) external signal <code>wdt_ext_timeout_r</code>. The time out signal ignores the input signal <code>wdt_ext_timeout_ack_r</code>; it operates purely as a status output signal. When asserted high, the signal can only be de-asserted by the writing to the respective clear bit in <code>EWDT_ESR</code> or clearing the flag bit in the respective <code>EWDT_CTRLx</code> register. It is the responsibility of the software to ensure that the <code>EWDT_ESR</code> register is updated appropriately in this trigger mode. A timer in this mode cannot exist with a timer set in mode 0x0.
F(Flag)	[3]	0	Indicates if a timeout error has occurred: 0x0: No error has occurred. 0x1: The watchdog timer counter has reached the value 0, an illegal window event. Clearing this bit once it has been set resets the interrupt, reset or external time out signal. Writing 1 to this register has no effect, and does not raise a timeout event.
RESERVED	[31:4]	0	Reserved. Read-as-zero; ignored on write.

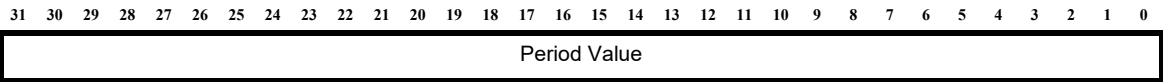
31.2.5 Watchdog Timer Period Register, EWDT_PERIODx

Address: 0xE04; Indexed for timer 0 to timer 3

Access: RW

Default: 0x0

Figure 31-5 EWDT_PERIODx Register



The watchdog timer period register stores the start value for the internal counter. This register can store a 32-bit or a 16-bit period value. The width of this register is configured at build time. Any unused bits in this register are read as zeros. When this register is written, the internal counter of the selected timer is automatically loaded with the value.

- You must unlock this register before writing to it. You can unlock the register by writing the correct pass code to the EWDT_PASSWD register. Otherwise, an illegal instruction exception is raised.
- After a write, this register is locked.

 **Note** You cannot use the AEX instruction to access this register.

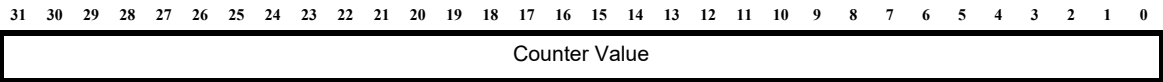
31.2.6 Watchdog Timer Counter Register, EWDT_COUNTx

Address: 0xE05; Indexed for timer 0 to timer 3

Access: R

Default: 0x0

Figure 31-6 EWDT_COUNTx Register



The watchdog timer count register provides the current value of the internal counter for the selected timer. When the cores are lockstepped (`cpu_safety == true`) and the watchdog is configured with a separate clock, (`watchdog_clk == true`), this register always returns the value 0 (zero).

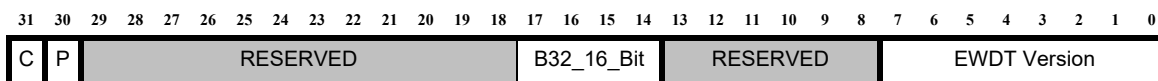
 **Note** You cannot use the AEX instruction to access this register.

31.2.7 Watchdog Timer Build Configuration Register, EWDT_BUILD

Address: 0xEE

Access: R

Figure 31-7 EWDT_BUILD Register



The register stores the watchdog configuration.



Note

When the windowed watchdog component is configured, the ERP_BUILD.WD field reflects the counter width of the first watchdog timer. The counter width of the other watchdog timers can be determined from the EWDT_BUILD register.

Table 31-6 EWDT_BCR Bit Definition

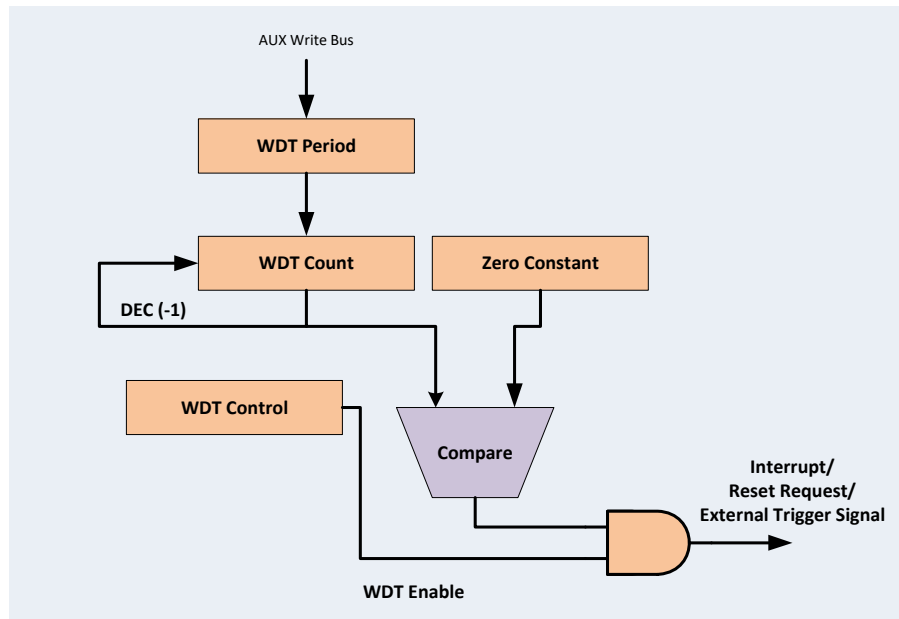
Field	Bits	Description
EWDT Version	[7:0]	0x1: This is the watchdog timer version number.
RESERVED	[13:8]	Reserved. Read-as-zero; ignored on write.
B32_16_Bit	[17:14]	Specifies the width of the counter in each watchdog timer. - bit [14]: when this bit is set to 1, it indicates that watchdog timer 0 is a 32-bit counter. A 0 indicates that the watchdog timer uses a 16-bit counter. - bit [15]: when this bit is set to 1, it indicates that watchdog timer 1 is a 32-bit counter. A 0 indicates that the watchdog timer uses a 16-bit counter. - bit [16]: when this bit is set to 1, it indicates that watchdog timer 2 is a 32-bit counter. A 0 indicates that the watchdog timer uses a 16-bit counter. - bit [17]: when this bit is set to 1, it indicates that watchdog timer 3 is a 32-bit counter. A 0 indicates that the watchdog timer uses a 16-bit counter.
RESERVED	[29:18]	Reserved. Read-as-zero; ignored on write.
P (Parity check enable)	[30]	When set to 1, all registers (except EWDT_PASSWD and EWDT_BUILD) within the timers are checked for parity errors.
C (Separate clock)	[31]	When set to 1, indicates a separate clock is applied to the watchdog timer.

31.3 Programming the WD Timer

31.3.1 Countdown Timer Mode

In countdown timer mode, the counter for the respective timer (when it is enabled) decrements from a fixed value that is set when the register `EWDT_PERIOD` is written. In countdown timer mode, a timeout error occurs when the timer's counter value reaches zero.

Figure 31-8 Timer Configured with No Window, Countdown Timer Mode



Programming countdown timer mode:

1. Select the timer you want to program in countdown timer mode.
 - a. Write `0x54` to `EWDT_PASSWD` register to unlock the index register `EWDT_INDEX` (if there is more than one timer in the build).
 - b. Write the timer number to be accessed to `EWDT_INDEX` register.
2. Specify the period value.
 - a. Write code `0x55` to `EWDT_PASSWD` register to unlock the period register (`EWDT_PERIOD`).
 - b. Write the period value to the `EWDT_PERIOD` register. The timer's counter is set to this value.
3. Enable timer in countdown mode and specify that an interrupt must be raised on a time out.
 - a. Write `0xAA` to `EWDT_PASSWD` register to unlock the control register (`EWDT_CTRL`).
 - b. Write `0x3` to the register to set up the timer in countdown mode and generate an interrupt on a time out.

The timer is enabled and the counter starts decrementing every clock cycle.

```
mov r0, 0x54
```

```

sr  r0, [EWDI_PASSWD]      ; unlock EWDI_INDEX for write access
mov r0, 0x3
sr  r0, [EWDI_INDEX]       ; select timer 3 for configuration
mov r0, 0x55
sr  r0, [EWDI_PASSWD]      ; unlock EWDI_PERIOD3 for write access
mov r0, 4000
sr  r0, [EWDI_PERIOD]      ; set the period reg (and counter) to 4000
mov r0, 0xAA
sr  r0, [EWDI_PASSWD]      ; unlock EWDI_CTRL3 for write access
mov r0, 0x3
sr  r0, [EWDI_CTRL]       ; enable timer; set interrupt time out

```

Preventing Time Out

In countdown mode, the application must ensure that the timer's counter never reaches the value zero. When a time out occurs, the timer raises an interrupt, initiate a system reset, or asserts the external time out signal. When the internal counter reaches the value zero the internal counter wraps to the value 32'hFFFF_FFFF for the 32-bit counter and 16'hFFFF for the 16-bit counter. The internal counter continues to decrement from this value until it is disabled (`EWDI_CTRLx[0] == 0`), the core halts or the counter is reset to a new value.

1. Select the timer you want to program in countdown timer mode.
 - a. Write 0x54 to EWDI_PASSWD register to unlock the index register EWDI_INDEX (if there is more than one timer in the build).
 - b. Write the timer number to be accessed to EWDI_INDEX register.
2. Reset the period value.
 - a. Write code 0x55 to EWDI_PASSWD register to unlock the period register (EWDI_PERIOD).
 - b. Write the period value in the EWDI_PERIOD register. The timer's counter is reset to this value.

```

reset_timer:
                                ; application code execution stops, reset time

mov r0, 0x55
sr  r0, [EWDI_PASSWD]         ; unlock EWDI_PERIOD3 for write access
mov r0, 4000
sr  r0, [EWDI_PERIOD]         ; set the period reg (and counter) to 4000
                                ; application code execution continues
jal [blink]                   ; return back to application code

```

32

Peripheral Bus

The peripheral region is an optional component that allows simple memory-mapped peripherals to be connected to the processor core, using a dedicated bus. The peripheral data bus can be configured independently from the memory bus protocol. The peripheral region can be changed at runtime through the DMP_PER_AUX register.

32.1 DMP Auxiliary Registers

Table 32-1 DMP Registers

Address	Auxiliary Register Name	Description
0x20A	DMP_PER_AUX	Peripheral memory region programming register
0xC3	DMP_PP_BUILD	Peripheral region build configuration register

32.1.1 Peripheral Memory Region, DMP_PER_AUX

Address: 0x20A

Access: RW

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Default Build-time configurable

Figure 32-1 DMP_PER_AUX, start of peripheral region



When at least one peripheral bus is enabled, the DMP_PER_AUX register exists, and identifies the mapping of the peripheral bus to the memory region. Memory regions mapped to peripheral bus are not cacheable.



Note

To ensure a consistent view of data, the data cache must be flushed whenever DMP_PER_AUX is modified. If data caches are not configured, no flush operations are required.



Caution

- Writing to this register alters the physical memory map, and may interfere with the outstanding memory operations. It is the responsibility of the software to synchronize all the outstanding memory operations before modifying this register.
- When different components are mapped to the same memory region, the access priority is as follows from the highest to the lowest:
 - ICCM0
 - ICCM1
 - DCCM
 - peripheral interface

Programming Notes:

- Programming the limit to zero effectively sets the region from base to the highest memory address.
- The base address of the region is inclusive, that is, the region includes the base address.
- The limit is exclusive, that is, the region excludes this address.

- Avoid programming base to a value greater or equal than the limit. If so programmed, the region spans from the base to the highest memory address.

Figure 32-2 Peripheral Regions Based on Base and Limit Values

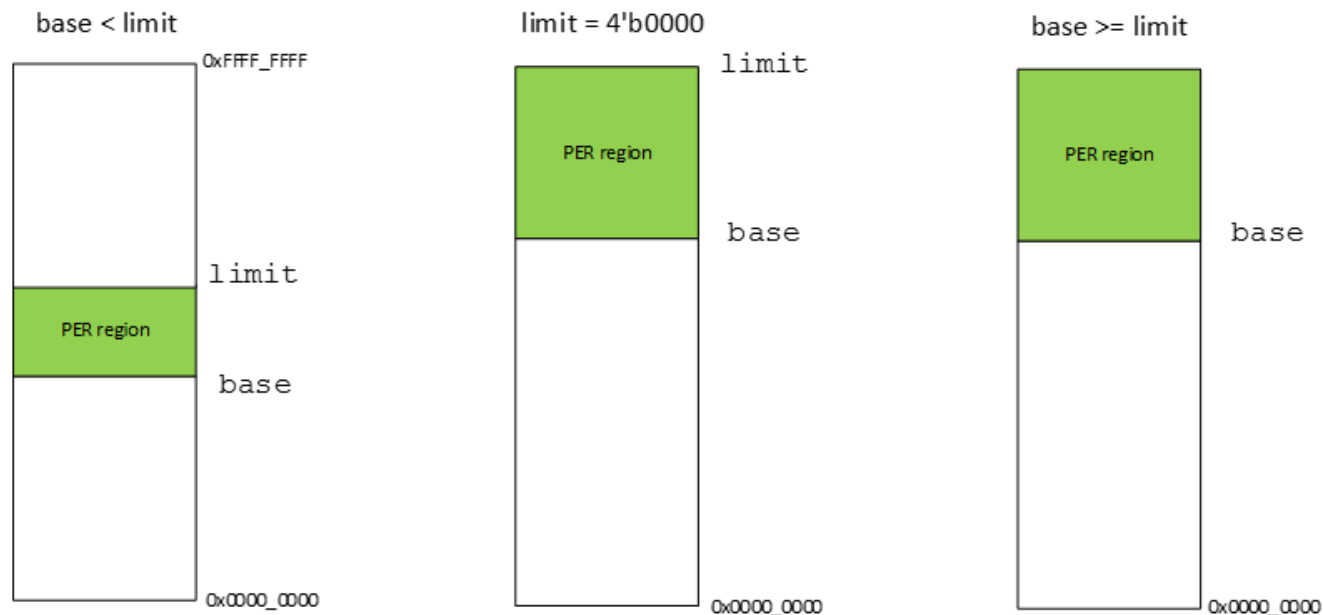


Table 32-2 DMP_PER_AUX Field Descriptions

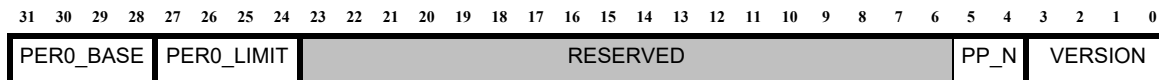
Field	Bit	Description
RESERVED	[26:0]	Reserved. Read-as-zero; ignored on write.
per0_limit	[27:24]	Specifies the end of the peripheral aperture.
per0_base	[31:28]	Specifies the memory region of the peripheral aperture.

32.1.2 Peripheral Region Build Configuration Register, DMP_PP_BUILD

Address: 0xC3

Access: R

Figure 32-3 DMP_PP_BUILD Register



This register indicates the presence of a peripheral region, and its build-time configuration.

Table 32-3 DMP_PP_BUILD Field Descriptions

Field	Bit	Description
VERSION	[3:0]	Version of the peripheral region component. All non-enumerated values are reserved. 0x2: current version
PP_N	[5:4]	Number of peripheral buses. All non-enumerated values are reserved. 0x0: One peripheral region. 0x1: Two peripheral regions.
RESERVED	[23:6]	Reserved. Read-as-zero; ignored on write.
PER0_LIMIT	[27:24]	Specifies the end of the peripheral aperture. This options defaults to the configured value of the <code>-dmp_per0_limit</code> option.
PER0_BASE	[31:28]	Specifies the memory region of the peripheral aperture. This options defaults to the configured value of the <code>-dmp_per0_base</code> option.

33

External Host Debugging

33.1 External Host Debugging Introduction

If the configuration includes the interface to an external debug host, additional registers are included in the auxiliary register set. This auxiliary register set includes at least the DEBUG and DEBUGI register, and may also include other registers to support, for example, real-time tracing capabilities. For specific details of advanced debug capabilities, refer to the databook for your ARCV2-based processor.

33.2 Host Debug Register Interface

Table 33-1 Host Debug Interface Auxiliary Registers

Address	Auxiliary Register Name	Description
0x5	Debug Register, DEBUG	Debug register
0x0F	Interrupt and Register Bank Debug Register, DEBUGI	Debug register, interrupts/register banks
0x30	Architectural Clock Gating Control Register, ACG_CTRL	Architectural clock gating

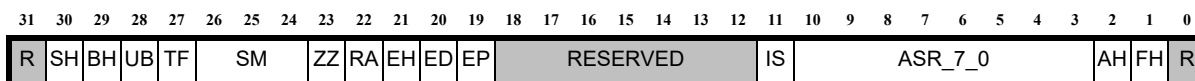
33.2.1 Debug Register, DEBUG

Address: 0x05

Access: RG

Reset: 0x0040_0000

Figure 33-1 DEBUG Register



The debug register is standard in all ARCV2-based processors. However, when actionpoints are enabled, the optional actionpoint system extends the DEBUG register (bits [10:2]) to give additional status information for the actionpoint mechanism. For more information about the format of the DEBUG register when actionpoints are enabled, see [“Actionpoint Extensions to Debug Register, DEBUG, 0x05”](#) on page 1043.

The debug register (DEBUG) contains the following bits:

Table 33-2 DEBUG Register Field Description

Field	Bit	Reset	Description
Force halt (FH)	[1]	0	This bit exists in configurations that include ARC Trace. FH is the approved method of stopping the ARCV2-based processor externally by the host. This bit does not have any affect if the ARCV2-based processor is already halted and the host sets this bit. The FH bit is not a mirror of the STATUS register H bit, that is, clearing FH does <i>not</i> start the processor. The FH bit always returns 0 when it is read.
Actionpoint Halt Flag (AH)	[2]	0	The actionpoint halt flag (bit 2), of the DEBUG register, indicates that an actionpoint has halted the processor when the halt flag is set to 1. The AH flag is cleared when the halt flag (H) in the STATUS32 register (see STATUS32) is cleared, for example, by restarting (see Starting) or single-stepping (see Single Instruction Stepping) the ARCV2-based processor. When actionpoints use a software exception, the AH bit to 0. Note: This bit is available only if actionpoints are enabled in the build (has_actionpoints==true). If actionpoints are disabled (has_actionpoints==false), this bit is read as zero and ignored on write.

Table 33-2 DEBUG Register Field Description

Field	Bit	Reset	Description
Actionpoints Status Register field (ASR_7_0)	[10:3]	0	The actionpoints Status Register field (bits [10:3]) of the DEBUG register, indicates which actionpoints have triggered an actionpoint event, independent of whether the actionpoint raises a software exception or a processor halt. Each bit, from ASR0 to ASR7, indicates which actionpoints from 0 to 7 have triggered the actionpoint event. When actionpoints are linked forming pairs or quads, only the master actionpoint of the pair or quad has its ASR bit set when the actionpoint is triggered. The ASR bits are sticky, and Actionpoints trigger only when their corresponding ASR bit transitions from 0 to 1. Therefore, when an Actionpoint condition is met and the ASR bit for the Actionpoint is set to 1, the exception or breakpoint associated with that Actionpoint is triggered only if ASR bit was previously 0. The corresponding ASR bit for an actionpoint is automatically cleared when the host or core writes to any of the actionpoint's associated control registers AP_AMVn, AP_AMMn, or AP_ACn (see Actionpoint Match Value, AP_AMV0, Actionpoint Match Mask, AP_AMM0, and Actionpoint Control, AP_AC0). Note: This field is available only if actionpoints are enabled in the build (has_actionpoints==true). If actionpoints are disabled (has_actionpoints==false), this field is read as zero and ignored on write.
Single instruction step (IS)	[11]	0	Single instruction stepping is provided through the use of the IS bit. If the host sets the IS bit, the ARCV2 core executes one instruction.
RESERVED	[18:12]	0	Reserved. Read-as-zero; ignored on write.
EP	[19]	0	This enable bit controls the gated clock driving the modules associated with the performance counters. 0x1: Enables the clock to the performance counters 0x0: Disables the clock to the performance counters Note: This bit is available only if performance counters are enabled in the build. If performance counters are disabled the bit is reserved.
ED	[20]	0	This enable bit controls the gated clock driving the modules associated with the actionpoints. 0x1: Enables the clock to the actionpoints module 0x0: Disables the clock to the actionpoints module Note: This bit is available only if actionpoints are configured in the build. If actionpoints are not configured the bit is reserved.

Table 33-2 DEBUG Register Field Description

Field	Bit	Reset	Description
External Halt (EH)	[21]	0	The EH bit is set when the processor is halted by external logic. The <code>arc_halt_req_a</code> signal is asserted by external logic that needs to halt the ARCV2-based processor. This signal may be asserted asynchronously with respect to the processor clock. When the processor is ready to halt, it enters the halted state in response to the assertion of <code>arc_halt_req_a</code> , sets the EH bit, and asserts the <code>arc_halt_ack</code> acknowledgment signal. This bit is a status bit, and the host cannot write to this bit.
Reset applied (RA)	[22]	1	RA bit is used by the debug host to determine that a target reset has occurred. The RA bit is set to 1 on a hard reset of the CPU and can be written only by the host.
Sleep state (ZZ)	[23]	0	ZZ bit indicates that the ARCV2-based processor is in the “sleep” state. Use the SLEEP instruction to force the ARCV2-based processor enter the sleep state. This bit is cleared whenever the ARCV2-based processor wakes from sleep state. This bit is a status bit, and the host cannot write to this bit.
Sleep Mode (SM)	[26:24]	0	SM field indicates the sleep mode of the processor. This field controls whether clocks for the internal timers and real-time clock are turned off during sleep state. For more information about the state of internal clocks during sleep state, see Table 33-3 . This bit is a status bit, and the host cannot write to this bit.
Triple Fault (TF)	[27]	0	When this bit is set to 1, it indicates that the processor has halted due to a triple fault. A triple fault occurs when an exception occurs during the fetching of an exception vector for an EV_MachineCheck (double fault) exception.
User-mode breakpoint (UB)	[28]	0	UB bit indicates that BRK is enabled in user mode. This bit is provided to allow an external debugger to debug user-mode tasks. This bit is always set to 0 to ensure that a user-mode task cannot stop the processor by executing a BRK instruction. If actionpoints are enabled, actionpoint halt in user mode requires that the DEBUG.UB bit is set. When transitioning from a kernel mode to user mode, use a SYNC instruction is required (DEBUG.UB cleared) to ensure that all kernel watchpoints are taken before transitioning to the user mode.
Breakpoint halt (BH)	[29]	0	BH bit is set when core is halted by execution of a breakpoint. This bit is cleared when the H bit in the STATUS32 register is cleared, that is, when single-stepping or restarting the ARCV2-based processor from the halted state.

Table 33-2 DEBUG Register Field Description

Field	Bit	Reset	Description
Self halt (SH)	[30]	0	SH indicates that the ARCV2-based processor has halted itself with the FLAG instruction. This bit is cleared whenever the H bit in the STATUS register is cleared, that is, the ARCV2-based processor is running or a single step has been executed.
RESERVED	[31]	0	Reserved. Read-as-zero; ignored on write.

Table 33-3 lists the state of various internal clocks during various sleep modes.

Table 33-3 Internal Clock During Sleep Mode

Sleep Mode (DEBUG.SM)	Core	Timers	Real-time Clock	IRQ Resync Clock
0	Disabled	Enabled	Enabled	Disabled unless required
1	Disabled	Disabled	Enabled	Disabled unless required
2	Disabled	Enabled	Disabled	Disabled unless required
3	Disabled	Disabled	Disabled	Disabled unless required
4	Disabled	Disabled	Disabled	Disabled unless required
5	Disabled	Disabled	Disabled	Disabled unless required
6	Disabled	Disabled	Disabled	Disabled unless required
7	Disabled	Disabled	Disabled	Disabled unless required

**Note**

If the core is configured with a watchdog timer, and the watchdog timer is not configured to be clocked with an external clock, the watchdog timer is also disabled for all sleep modes except sleep mode 0 and 2.

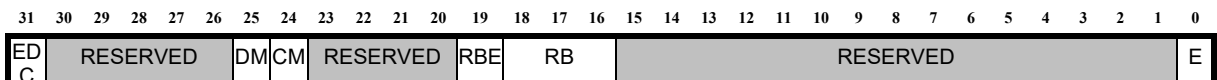
33.2.2 Interrupt and Register Bank Debug Register, DEBUGI

Address: 0x0F

Access: RG

Reset: 0x0000_0000

Figure 33-2 DEBUGI Register



This register allows the debug host an independent selection of which register bank to access when accessing core registers.

You can also use this register through a debug host to enable the exception or interrupt vector break functionality.

Table 33-4 DEBUGI Register Field Description

Field	Bit	Description
E	[0]	When set to 1, the exception or interrupt vector break functionality is enabled. In this mode, on an interrupt or an exception, the processor executes a breakpoint at the location of the corresponding vector in the exception table if the LSB bit of the vector value is set to 1. As instruction addresses are always 16 bit aligned, this bit is unused for addressing purposes and functions only to enable a breakpoint at this vector. The breakpoint takes affect in place of the final jump of the sequence. So, the entire sequence is replayed before the breakpoint is executed.
RESERVED	[15:1]	Reserved. Read-as-zero; ignored on write.
RB	[18:16]	Register Bank (present with <code>RGF_NUM_BANKS > 1</code>) Pre-empts STATUS32.RB for all instructions originating from the debug interface when DEBUGI.RBE is set. When DEBUGI.RBE is not set, the existing value STATUS32.RB is used where RBE = bit 19. If <code>RGF_NUM_BANKS {== 1, < 2, < 4}</code> , the {RB, RB[2:1], RB[2]} field is read as zero and ignored on write.
RBE	[19]	Register Bank Enable. When this bit is set, DEBUGI.RB value preempts STATUS32.RB for all instructions originating from the debug interface. When DEBUGI.RBE is not set, the existing value STATUS32.RB is used.
RESERVED	[23:20]	Reserved. Read-as-zero; ignored on write.

Table 33-4 DEBUGI Register Field Description

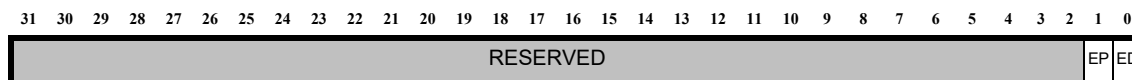
Field	Bit	Description
CM	[24]	Indicates the operating mode of the core ■ 0x0: normal mode ■ 0x1: secure mode This bit is valid only when <code>-sec_modes_option==true</code>. Otherwise, this bit is read as zero and ignored on writes.
DM	[25]	Indicates the operating mode of the debug port ■ 0x0: normal mode ■ 0x1: secure mode This bit is valid only when <code>-sec_modes_option==true</code>. Otherwise, this bit is read as zero and ignored on writes.
RESERVED	[30:26]	Reserved. Read-as-zero; ignored on write.
EDC	[31]	For ARC EM4-based processors, this bit is reserved. ■ 0x1: Memory accesses by the debugger are encrypted. ■ 0x0: Memory accesses by the debugger are not encrypted.

33.2.3 Architectural Clock Gating Control Register, ACG_CTRL

Address: 0x30

Access: RW

Figure 33-3 ACG_CTRL Register



Use this register to enable or disable the actionpoint and the performance monitor clocks in the kernel mode by software or in the debug mode through host access. This register is present only if your processor build is configured with architectural clock gating and has actionpoints or performance counters (`has_actionpoints==true` or `has_pct==true`).

Table 33-5 ACG_CTRL Field Description

Field	Bit	Description
ED	[0]	Enable or disable the actionpoint clock in the kernel mode by the software or in the debug mode through host access. User mode access to this bit generates a privilege violation exception. Set this bit to 1 to enable the actionpoint clock. If this bit is set to 0, the actionpoint clock is gated. This bit has the same functionality as the <code>DEBUG[20]</code> bit, but with the extra functionality that <code>ACG_CTRL[ED]</code> is also accessible in kernel mode by the software.
EP	[1]	Enable performance monitor clock in the kernel mode by the software or in the debug mode through host access. User mode access to this bit generates a privilege violation exception. Set this bit to 1 to enable the performance monitor clock. If this bit is set to 0, the performance monitor clock is gated. This bit has the same functionality as the <code>DEBUG[19]</code> bit, but with the extra functionality that <code>ACG_CTRL[EP]</code> is also accessible in kernel mode by the software.

34

Debug Features

34.1 Debug Introduction

This chapter describes the debug features of the ARCV2-based processor series. These features include the following:

- Actionpoints
- SmaRT – Small Real-Time Trace Buffer

34.2 Actionpoints

The ARCV2 ISA provides an optional actionpoint system for enhanced debugging of application software in real time. This section describes the actionpoint registers; for information about using the actionpoint system see the Databook for your processor series.

The actionpoints mechanism supports breakpoints and watchpoints. A *breakpoint* is an actionpoint that triggers on a specific instruction, or range of instructions, being executed. A *watchpoint* is an actionpoint that detects specific address or data values being accessed by memory-referencing instructions, auxiliary register read (LR) and write (SR) instructions, or the presence of specific values on the two optional external parameter ports.

Both breakpoints and watchpoints can be programmed to either halt the processor or raise an exception. Certain actionpoints take effect before the triggering instruction completes (pre-commit), while others take effect after the triggering instruction completes (post-commit). In ARC EM family processors, only the actionpoints that trigger on memory/auxiliary address and memory/auxiliary data values are post-commit; that is, they take effect after the triggering instruction completes. For more information about triggering delays, see the databook for your processor series.

34.2.1 Actionpoint Auxiliary Registers

Actionpoints are programmed using a collection of control and status registers mapped into the auxiliary register space.

Table 34-1 Auxiliary Registers for Actionpoints

Auxiliary Register Address	Register Name	Description
0x05	DEBUG	Extensions to Debug Register; (see Actionpoint Extensions to Debug Register, DEBUG, 0x05)
0x76	AP_BUILD	Actionpoint build configuration register(see Actionpoints Configuration Register, AP_BUILD)
0x220	AP_AMV0	Actionpoint 0 Match Value (see Actionpoint Match Value, AP_AMV0)
0x221	AP_AMM0	Actionpoint 0 Match Mask (see Actionpoint Match Mask, AP_AMM0)
0x222	AP_AC0	Actionpoint 0 Control (see Actionpoint Control, AP_AC0)
0x223	AP_AMV1	Actionpoint 1 Match Value (see Actionpoint Match Value, AP_AMV1)
0x224	AP_AMM1	Actionpoint 1 Match Mask (see Actionpoint Match Mask, AP_AMM1)
0x225	AP_AC1	Actionpoint 1 Control (see Actionpoint Control, AP_AC1)
0x226	AP_AMV2	Actionpoint 2 Match Value (see Actionpoint Match Value, AP_AMV2)
0x227	AP_AMM2	Actionpoint 2 Match Mask (see Actionpoint Match Mask, AP_AMM2)
0x228	AP_AC2	Actionpoint 2 Control (see Actionpoint Control, AP_AC2)
0x229	AP_AMV3	Actionpoint 3 Match Value (see Actionpoint Match Value, AP_AMV3)
0x22A	AP_AMM3	Actionpoint 3 Match Mask (see Actionpoint Match Mask, AP_AMM3)
0x22B	AP_AC3	Actionpoint 3 Control (see Actionpoint Control, AP_AC3)
0x22C	AP_AMV4	Actionpoint 4 Match Value (see Actionpoint Match Value, AP_AMV4)
0x22D	AP_AMM4	Actionpoint 4 Match Mask (see Actionpoint Match Mask, AP_AMM4)
0x22E	AP_AC4	Actionpoint 4 Control (see Actionpoint Control, AP_AC4)
0x22F	AP_AMV5	Actionpoint 5 Match Value (see Actionpoint Match Value, AP_AMV5)
0x230	AP_AMM5	Actionpoint 5 Match Mask (see Actionpoint Match Mask, AP_AMM5)
0x231	AP_AC5	Actionpoint 5 Control (see Actionpoint Control, AP_AC5)

Table 34-1 Auxiliary Registers for Actionpoints (Continued)

Auxiliary Register Address	Register Name	Description
0x232	AP_AMV6	Actionpoint 6 Match Value (see Actionpoint Match Value, AP_AMV6)
0x233	AP_AMM6	Actionpoint 6 Match Mask (see Actionpoint Match Mask, AP_AMM6)
0x234	AP_AC6	Actionpoint 6 Control (see Actionpoint Control, AP_AC6)
0x235	AP_AMV7	Actionpoint 7 Match Value (see Actionpoint Match Value, AP_AMV7)
0x236	AP_AMM7	Actionpoint 7 Match Mask (see Actionpoint Match Mask, AP_AMM7)
0x237	AP_AC7	Actionpoint 7 Control (see Actionpoint Control, AP_AC7)
0x23E	AP_ISO_EN	Actionpoint IsolationEnable (see Actionpoint Isolation Enable Register, AP_ISO_EN)
0x23F	AP_WP_PC	Watchpoint Program Counter (see Watchpoint Program Counter, AP_WP_PC)
0x30	ACG_CTRL	Register to enable or disable the actionpoint clocks.

**Note**

For more details on the access code definitions of the registers, see [Table 5-7 – “Key to Auxiliary Register Access Permissions for LR and SR Instructions”](#) on page 143.

34.2.1.1 Access Requirements

All actionpoint registers are accessible only in kernel mode; accessing these registers in user mode raises a Privilege Violation exception.

Vector Name	Vector Offset	Exception Cause Register
EV_PrivilegeV	0x1C	0x0700nn

The parameter field (nn) gives the number of the action point that triggered the exception.

- 0x00 = AP0
- 0x01 = AP1
- 0x02 = AP2
- 0x03 = AP3
- 0x04 = AP4

- 0x05 = AP5
- 0x06 = AP6
- 0x07 = AP7

If there are fewer than eight actionpoints, only the associated registers exist and accessing non-existent actionpoint registers raises an Illegal Instruction Error exception in both kernel and user modes.

Vector Name	Vector Offset	Exception Cause Register
Instruction Error	0x08	0x020000

34.2.1.2 Actionpoint Registers Host and Core Access

When the core is running, the enabled actionpoints must be reconfigured carefully because it may result in undefined behavior. The following is the recommended procedure:

- The host must first halt the core
- The host then checks the ASR status field and AH bit of the DEBUG register (see [Actionpoint Extensions to Debug Register, DEBUG, 0x05](#)) to see whether an actionpoint triggered during the halt, and takes appropriate action if necessary.

After these actions, the host can reconfigure actionpoints, and then restart the processor.

34.2.1.3 Actionpoint Registers Reset State

[Table 34-2](#) lists the values of the actionpoint registers when the ARCV2-based processor is reset.

Table 34-2 State of Auxiliary Registers upon Reset

Register Name	Initial Reset Values
DEBUG	0x00000000 for 2, 4 or 8 actionpoints
AP_AMV0 to AP_AMV7	0x00000000 for 2, 4 or 8 actionpoints
AP_AMM0 to AP_AMM7	0x00000000 for 2, 4 or 8 actionpoints
AP_AC0 to AP_AC7	0x00000000 for 2, 4 or 8 actionpoints
AP_BUILD	0x00000005 = 2 actionpoints, full targets 0x00000105 = 4 actionpoints, full targets 0x00000205 = 8 actionpoints, full targets 0x00000405 = 2 actionpoints, minimum targets 0x00000505 = 4 actionpoints, minimum targets 0x00000605 = 8 actionpoints, minimum targets

Table 34-2 State of Auxiliary Registers upon Reset (Continued)

Register Name	Initial Reset Values
AP_WP_PC	0x00000000

34.2.2 Actionpoint Extensions to Debug Register, DEBUG, 0x05

The debug register is standard in all ARCV2-based processors. The optional actionpoint system extends the [Debug Register, DEBUG](#) register (bits [10:2]) give additional status information for the actionpoint mechanism.

Each of the extension actionpoint fields in the DEBUG register performs the following functions:

- **AH:** The actionpoint halt flag (bit 2), of the DEBUG register, indicates that an actionpoint has halted the processor when the halt flag is set to 1. The AH flag is cleared when the halt flag (H) in the STATUS32 register (see [Status Register, STATUS32](#)) is cleared, for example, by restarting (see [Starting](#)) or single-stepping (see [Single Instruction Stepping](#)) the ARCV2-based processor. actionpoints that cause a software exception set the AH bit to 0.
- **ASR:** The actionpoints Status Register field (bits [10:3]) of the DEBUG register, indicates which actionpoints have triggered an actionpoint event, independent of whether the actionpoint raises a software exception or a processor halt. Each bit, from ASR0 to ASR7, indicates which actionpoints from 0 to 7 have triggered the actionpoint event. When actionpoints are linked forming pairs or quads, only the master actionpoint of the pair or quad has its ASR bit set when the actionpoint is triggered. The ASR bits are sticky, and Actionpoints trigger only when their corresponding ASR bit transitions from 0 to 1. Therefore, when an actionpoint condition is met and the ASR bit for the actionpoint is set to 1, the exception or breakpoint associated with that actionpoint is triggered only if ASR bit was previously 0. The ASR bit for an actionpoint is automatically cleared when the host or core writes to any of the actionpoint's associated control registers AP_AMVn, AP_AMMn, or AP_ACn (see [Actionpoint Match Value, AP_AMV0](#), [Actionpoint Match Mask, AP_AMM0](#), and [Actionpoint Control, AP_AC0](#)).

34.2.3 Action Points and Architectural Clock Gating

For runtimes that use architectural clock gating, use the `ACG_CTRL.ED` bit to enable or disable the actionpoint clock in the kernel mode by the software or in the debug mode through host access. User mode access to this bit generates a privilege violation exception.

For more information, see [“Architectural Clock Gating Control Register, ACG_CTRL”](#) on page 1038.

34.2.4 Actionpoints Configuration Register, AP_BUILD

Address: 0x76

Access: R

The `AP_BUILD` register (`AP_BUILD`) indicates the presence of the optional [“Actionpoints”](#) mechanism. The `AP_BUILD` register returns 0 when actionpoints are not available.

Figure 34-1 Actionpoint Build Register

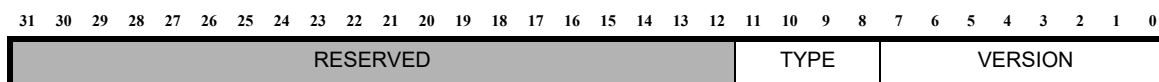


Table 34-3 Actionpoint Build Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Actionpoint Version ■ 0x5: Version of the actionpoint system ■ 0x7: Supports isolation of actionpoints
TYPE	[11:8]	Actionpoint Build Type Refers to the number of actionpoints selected in the ARCV2-based system: ■ 0x0: 2 Actionpoints, full set of targets ■ 0x1: 4 Actionpoints, full set of targets ■ 0x2: 8 Actionpoints, full set of targets ■ 0x4: 2 Actionpoints, minimum set of targets ■ 0x5: 4 Actionpoints, minimum set of targets ■ 0x6: 8 Actionpoints, minimum set of targets ■ 0x3, 0x7 to 0xF: Reserved
RESERVED	[31:12]	Reserved, read as zero and ignored on writes

34.2.5 Actionpoint Match Value, AP_AMV0

Address: 0x220

Access: RW

Reset: 0x0000_0000

Figure 34-2 Actionpoint 0 Match Value Register



This register, in conjunction with the Actionpoint Match Mask register, specifies the condition that must be met for the associated actionpoint to trigger. This register contains the value that the monitored target must match.

After an actionpoint is triggered, the value in AP_AMV0 is modified to contain the value of the target parameter that caused the trigger. This is used by a debugger to determine the exact value that caused the trigger when the triggering condition specifies a range of addresses or data values.

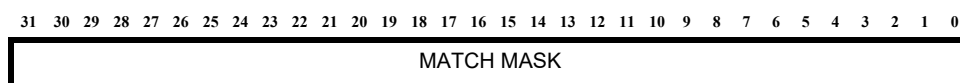
This register is set to 0x00000000 on reset.

You can read and write to this register.

34.2.6 Actionpoint Match Mask, AP_AMM0

Address: 0x221
 Access: RW
 Reset: 0x0000_0000

Figure 34-3 Actionpoint 0 Match Mask Register



This register contains the mask that is applied to both the monitored parameter and the match value (AP_AMV0) before the equality check is done.

At each bit position where a 1 is present, that bit plays *no role* in the comparison.

This register is set to 0x00000000 on reset.

This register is a read and write register.

For example, the following values specify a range of values from 0x12345000 to 0x12345FFF:

AP_AMV0 = 0x12345678

AP_AMM0 = 0x00000FFF

Depending on the setting of the Mode (M) bit in the Actionpoint Control register, the actionpoint triggers when the target value is within this range, or outside this range.

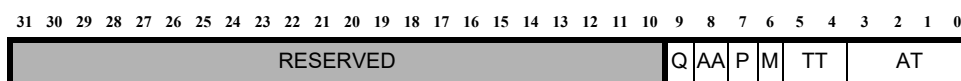
34.2.7 Actionpoint Control, AP_AC0

Address: 0x222

Access: RW

Reset: 0x0000_0000

Figure 34-4 Actionpoint 0 Control Register



Sets the target and operation mode.

This register is set to 0x00000000 on reset.

This register is a read and write register.

34.2.7.1 Actionpoint Target Field, AT

The Actionpoint Target (AT) field (bits [3:0]) of the AP_AC0 register defines which target to monitor with reference to the values in the Actionpoint Match Value (AP_AMV0) and Actionpoint Match Mask (AP_AMM0) registers. [Table 34-4](#) lists the selections available.

Table 34-4 Actionpoint Target Field, AT, Type

Actionpoint Type (AT)	Type of Data Comparison	Description
0x0	Instruction address	Instruction address trigger on specified PC value.
0x1	Instruction data	Instruction data trigger on specified instruction data.
0x2	Memory address	Memory address Trigger on access to specific memory address, from processor core only.
0x3	Memory data	Memory data trigger on a data value written to, or read from memory, from the processor core only. If both read and write are selected (TT=11) there is an implicit selection of write data only.
0x4	Auxiliary register address	Auxiliary register address trigger on access to specific auxiliary register address.
0x5	Auxiliary register data	Auxiliary register data trigger on data value written to, or read from an auxiliary register from the core only.
0x6	External parameter 0	External parameter 0 trigger on a specific value.
0x7	External parameter 1	External parameter 1 trigger on a specific value.
0x8 to 0xF	Reserved	N/A

**Caution**

When tracking accesses to the auxiliary registers (AT = 0100 or AT = 0101), only loads (LR) and stores (SR) to auxiliary registers can be monitored. Therefore, the STATUS32 register cannot be tracked when it is modified by the processor, that is, processor generated PC updates to the STATUS32 register.

34.2.7.2 Transaction Type Field, TT

The Transaction Type (TT) field (bits [5:4]) of the AP_AC0 register defines what type of transaction is monitored. [Table 34-5](#) lists the possible values: none (disabled), read, write, read or write.

Table 34-5 Transaction Type Field, TT, Type

Transaction Type (TT)	Description
0x0	Disable the associated actionpoint.
0x1	Trigger when the transaction is a write and the target value matches.
0x2	Trigger when the transaction is a read and the target value matches.
0x3	Trigger when the transaction is a write or read and the target value matches. If the write and read match conditions occur concurrently, the write values are implicitly selected for a match. If the write values do not match, the read values are not considered for a match.

34.2.7.3 Mode Field, M

The Mode (M) field (bit 6) of the AP_AC0 register selects whether an actionpoint triggers when the target value is *inside* the match range produced by the associated AP_AMV0 and AP_AMM0 registers, or whether it triggers when the value is *outside* this range, as shown in [Table 34-6](#).

Table 34-6 Mode Field, M, Type

Mode (M)	Description
0x0	Trigger when in range
0x1	Trigger when outside range

34.2.7.4 Paired Field, P

The Pair (P) field (bit 7) of the AP_AC0 register, selects whether this actionpoint is a paired actionpoint. If an actionpoint is paired, both this actionpoint and the next one must meet their match conditions at the same time to cause a trigger (processor halt or software exception).

When actionpoint n (the *master* actionpoint in an actionpoint pair) is set to pair mode, it is combined with actionpoint $n+1$, to determine when an event occurs. If the P bit of the highest numbered actionpoint is set, it pairs with actionpoint 0. The possible pairings (for eight actionpoints) are therefore:

- (0,1)
- (1,2)
- (2,3)
- (3,4)
- (4,5)
- (5,6)
- (6,7)
- (7,0)

When the P bit for actionpoint n is set, the actionpoint $n+1$ is prevented from triggering on its own.

The P and Q bits of all actionpoints must be programmed such that there is no overlap in the actionpoint groups (pairs or quads) created. No actionpoint should belong to more than one such group. If such overlap is programmed, the triggering behavior of these groups is undefined.

When a paired actionpoint triggers, both actionpoint n and actionpoint $n+1$ have their respective AP_AMV registers updated with the appropriate target value.

The DEBUG ASR field and the ECR *nn* field only indicate the master of a pair that hits, that is, the actionpoint that has its P bit set. For example, if actionpoint 0 has its P bit set and an event occurs that matches the conditions of both actionpoint 0 and actionpoint 1, only bit 0 of the ASR is set.

A typical use of pairs is to detect writing of a particular value to a particular address or to more tightly specify a range of values than is possible to achieve with the Actionpoint Match Mask register of a single actionpoint.



Note

A pair containing one pre-commit actionpoint and one post-commit actionpoint never triggers because the former take effect before the triggering instruction completes; whereas the latter take effect after the triggering instruction completes. Therefore, the two triggering events never coincide.

Table 34-7 Pair Field, P, Type

Pair (P)	Description
0	Normal unpaired operation
1	Pair this actionpoint with the next actionpoint

34.2.7.5 Actionpoint Action Field, AA

The Actionpoint Action (AA) field (bit 8) of the AP_AC0 register defines whether the actionpoint causes the processor to halt, or raise an exception. Actionpoint halt in user mode requires that the `DEBUG.UB` bit is set.

Table 34-8 Actionpoint Action Field, AA Field, Type

Actionpoint Action (AA)	Description
0x0	Halt the processor If the processor is halted because of a Breakpoint, the instruction that caused the Breakpoint is not yet executed, and the PC continued to point at the triggering instruction. Watchpoints can halt the processor some cycles after the triggering instruction has been executed. In such case, the processor must be restarted from where it halted.
0x1	Raise an exception Cause a Privilege Violation exception (see Privilege Violation, Action-Point Hit Instruction Fetch). This exception causes the program flow to be redirected through the EV_PrivilegeV vector. The Cause Code and Parameter field of the ECR indicate whether an Actionpoint was triggered and which Actionpoint caused the exception. A pre-commit Actionpoint exception is raised before the triggering instruction is executed, whereas a post-commit Actionpoint exception is raised after the triggering instruction has executed. Actionpoints in this mode are not triggered when the AE bit of the STATUS32 register is set. This means that they do not cause a double exception leading to a machine check.

34.2.7.6 Quad Field, Q

The Quad (Q) field (bit 9) of the AP_AC0 register selects whether this actionpoint is a quad actionpoint. When set, this bit means that actionpoint n (the *master* actionpoint in an actionpoint quad) is set to quad mode and this actionpoint is combined with actionpoint $n+1$, actionpoint $n+2$, and actionpoint $n+3$ to determine when an event occurs. This is analogous to a paired actionpoint, that is, if an actionpoint has its Q bit set, this actionpoint and the next three actionpoints must all match their conditions for it to trigger. As with pairs, the actionpoints are selected using modulo number_of_actionpoints.

Quad actionpoints take precedence over paired actionpoints.

The possible quads for eight actionpoints are:

- (0,1,2,3)
- (1,2,3,4)
- (2,3,4,5)
- (3,4,5,6)
- (4,5,6,7)
- (5,6,7,0)

- (6,7,0,1)
- (7,0,1,2)

If an actionpoint n has its Q bit set, the actionpoints $n+1$, $n+2$, and $n+3$ are not triggered on their own. The P and Q bits of all actionpoints must be programmed such that there is no overlap in the actionpoint groups (pairs or quads) created. No actionpoints should belong to more than one such group. If such overlap is programmed, the triggering behavior of these groups is undefined.

The DEBUG ASR field and the ECR mn field only indicate the master of a quad that hits, that is, the actionpoint that has its Q bit set.

When a quad actionpoint triggers, all four actionpoints, n through actionpoint $n+3$, have their respective AP_AMV registers updated with the appropriate target value.

If the quad bit is set on an actionpoint in a build with only two actionpoints, it is never triggered.



Note

A quad containing a mixture of Breakpoints and Watchpoints never triggers because Breakpoints take effect *before* the triggering instruction completes, whereas Watchpoints take effect *after* the triggering instruction completes. Therefore, at least two of the four triggering events can never coincide.

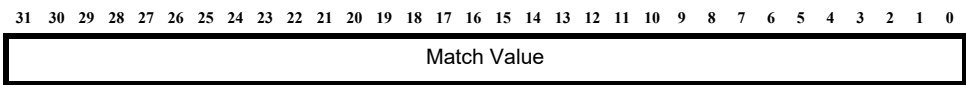
Table 34-9 Quad Type

Quad (Q)	Description
0x0	Normal operation
0x1	Quad this actionpoint with the next three actionpoints (modulo the number of actionpoints)

34.2.8 Actionpoint Match Value, AP_AMV1

Address: 0x223
Access: RW
Reset: 0x0000_0000

Figure 34-5 Actionpoint 1 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

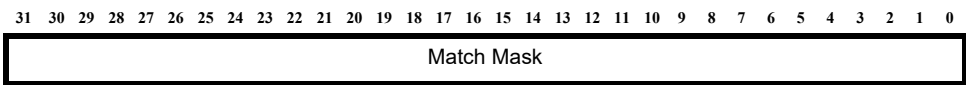
34.2.9 Actionpoint Match Mask, AP_AMM1

Address: 0x224

Access: RW

Reset: 0x0000_0000

Figure 34-6 Actionpoint 1 Match Mask Register

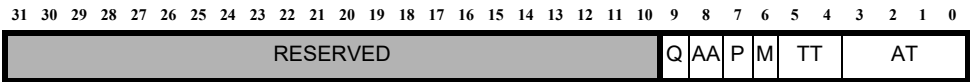


See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

34.2.10 Actionpoint Control, AP_AC1

Address: 0x225
Access: RW
Reset: 0x0000_0000

Figure 34-7 Actionpoint 1 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

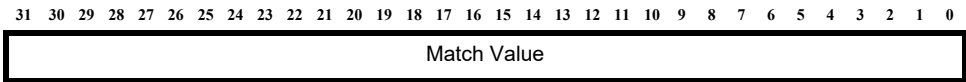
34.2.11 Actionpoint Match Value, AP_AMV2

Address: 0x226

Access: RW

Reset: 0x0000_0000

Figure 34-8 Actionpoint 2 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

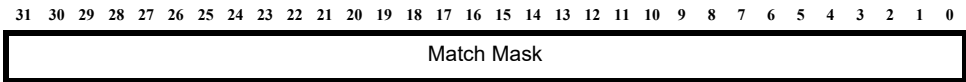
34.2.12 Actionpoint Match Mask, AP_AMM2

Address: 0x227

Access: RW

Reset: 0x0000_0000

Figure 34-9 Actionpoint 2 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

34.2.13 Actionpoint Control, AP_AC2

Address: 0x228

Access: RW

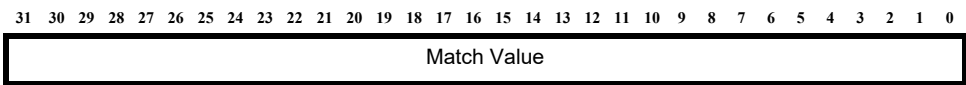
Reset: 0x0000_0000

See [Actionpoint Control, AP_AC0](#) register for field information.

34.2.14 Actionpoint Match Value, AP_AMV3

Address: 0x229
Access: RW
Reset: 0x0000_0000

Figure 34-10 Actionpoint 3 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

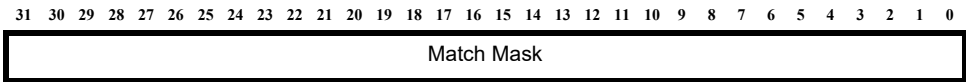
34.2.15 Actionpoint Match Mask, AP_AMM3

Address: 0x22A

Access: RW

Reset: 0x0000_0000

Figure 34-11 Actionpoint 3 Match Mask Register

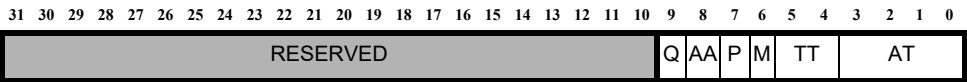


See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

34.2.16 Actionpoint Control, AP_AC3

Address: 0x22B
Access: RW
Reset: 0x0000_0000

Figure 34-12 Actionpoint 3 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

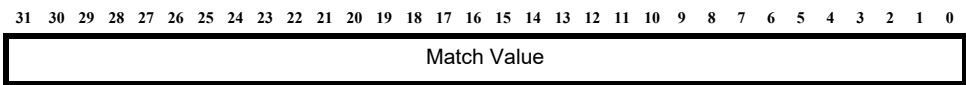
34.2.17 Actionpoint Match Value, AP_AMV4

Address: 0x22C

Access: RW

Reset: 0x0000_0000

Figure 34-13 Actionpoint 4 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

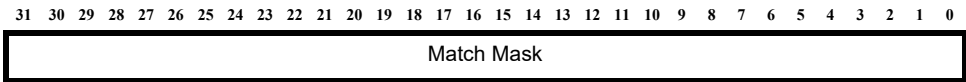
34.2.18 Actionpoint Match Mask, AP_AMM4

Address: 0x22D

Access: RW

Reset: 0x0000_0000

Figure 34-14 Actionpoint 4 Match Mask Register

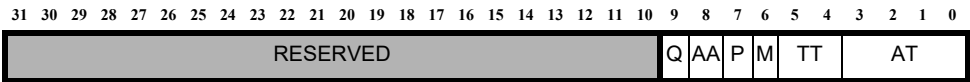


See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

34.2.19 Actionpoint Control, AP_AC4

Address: 0x22E
Access: RW
Reset: 0x0000_0000

Figure 34-15 Actionpoint 4 Control Register

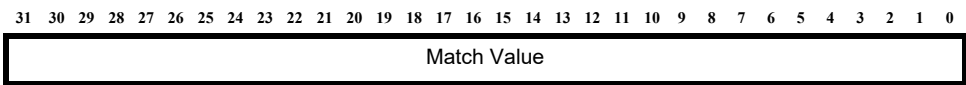


See [Actionpoint Control, AP_AC0](#) register for field information.

34.2.20 Actionpoint Match Value, AP_AMV5

Address: 0x22F
Access: RW
Reset: 0x0000_0000

Figure 34-16 Actionpoint 5 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

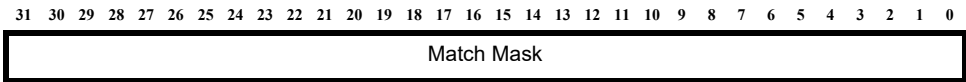
34.2.21 Actionpoint Match Mask, AP_AMM5

Address: 0x230

Access: RW

Reset: 0x0000_0000

Figure 34-17 Actionpoint 5 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

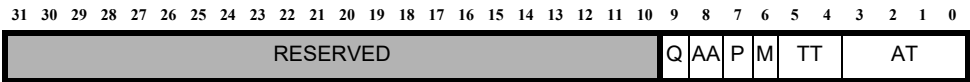
34.2.22 Actionpoint Control, AP_AC5

Address: 0x231

Access: RW

Reset: 0x0000_0000

Figure 34-18 Actionpoint 5 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

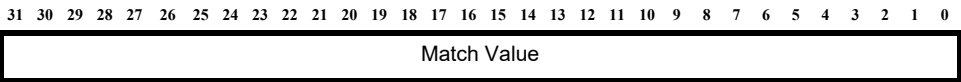
34.2.23 Actionpoint Match Value, AP_AMV6

Address: 0x232

Access: RW

Reset: 0x0000_0000

Figure 34-19 Actionpoint 6 Match Value Register

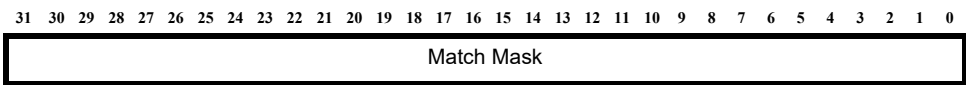


See [Actionpoint Match Value, AP_AMV0](#) register for field information.

34.2.24 Actionpoint Match Mask, AP_AMM6

Address: 0x233
Access: RW
Reset: 0x0000_0000

Figure 34-20 Actionpoint 6 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

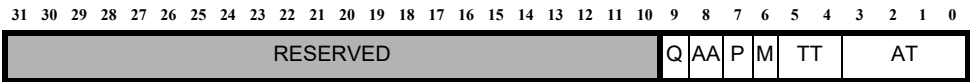
34.2.25 Actionpoint Control, AP_AC6

Address: 0x234

Access: RW

Reset: 0x0000_0000

Figure 34-21 Actionpoint 6 Control Register

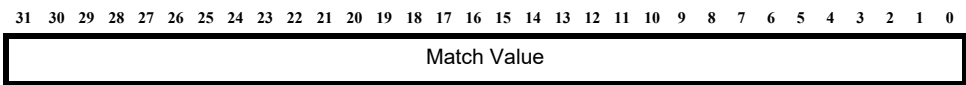


See [Actionpoint Control, AP_AC0](#) register for field information.

34.2.26 Actionpoint Match Value, AP_AMV7

Address: 0x235
Access: RW
Reset: 0x0000_0000

Figure 34-22 Actionpoint 7 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

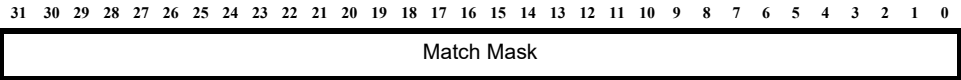
34.2.27 Actionpoint Match Mask, AP_AMM7

Address: 0x236

Access: RW

Reset: 0x0000_0000

Figure 34-23 Actionpoint 7 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

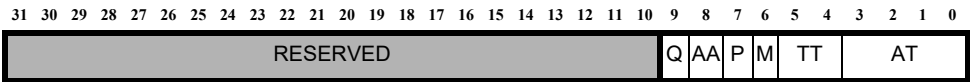
34.2.28 Actionpoint Control, AP_AC7

Address: 0x237

Access: RW

Reset: 0x0000_0000

Figure 34-24 Actionpoint 7 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

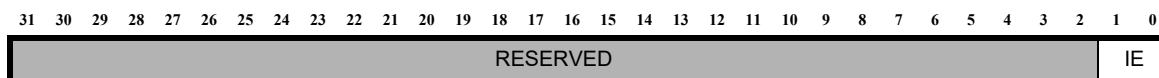
34.2.29 Actionpoint Isolation Enable Register, AP_ISO_EN

Address: 0x23E

Access: RW

Reset: 0x1

Figure 34-25 Actionpoint Isolation Enable Register



Use this register to enable or disable actionpoint isolation.

Table 34-10 Actionpoint Isolation Enable Field Descriptions

Field	Bit	Reset	Description
IE	[1:0]	1	Isolation Enable (IE) field (bits 1:0) of the AP_ISO_EN register defines whether the isolation of action points is to be enabled or not. 0x0, 0x3: Invalid values; Programming these values generates parity error, which is aggregated in the safety monitor to generate <code>emsi_error</code> in builds where safety manager is present. Programming these values will be ignored, when safety manager is not present. 0x1: Disable isolation of actionpoints (default) 0x2: Enable isolation of actionpoints It is read as 0x1, when isolation is not enabled through this register field AND through the <code>secure_aps_iso_en</code> input signal. It is read as 0x2, when isolation is enabled either through this register field OR through the <code>secure_aps_iso_en</code> input signal. This register is not updated with 0x0 and 0x3.
RESERVED	[31:2]	0	Reserved, read as zero and ignored on writes

34.2.30 Watchpoint Program Counter, AP_WP_PC

Address: 0x23F
 Access: R
 Reset: 0x0000_0000

Figure 34-26 AP_WP_PC, Watchpoint Program Counter Register when PC_SIZE = 32

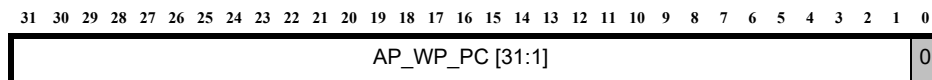
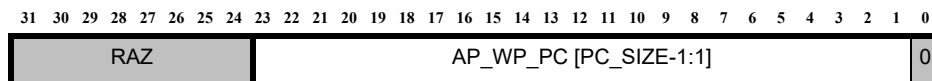


Figure 34-27 AP_WP_PC Register when PC_SIZE < 32



AP_WP_PC is a read-only register.

An ARCV2-based processor may execute one or two instructions after a watchpoint is triggered, because of the implementation-specific delay required to detect the watchpoint. It then either halts the processor or raises a Privilege Violation exception, depending on the Actionpoint Action (AA) bit of the triggered actionpoint.

To assist the debugger in identifying the location of the watchpoint-triggering instruction, the AP_WP_PC register contains the program address of the last instruction to trigger a watchpoint exception or halts.

Actionpoints on external parameters do not have an associated PC and for these actionpoints, the AP_WP_PC is undefined. Also, any graduating instruction data may complete out of order and a watchpoint triggered on such a instruction data does not have the instruction PC loaded in the AP_WP_PC register.

34.3 SmaRT

Small real time trace (SmaRT) is an optional on-chip debug hardware component that captures instruction-trace history.

SmaRT stores the address of the most recent non-sequential instructions executed.

The MetaWare debugger enables SmaRT and displays the instruction trace history. The MetaWare debugger reads the saved instruction trace information via the JTAG port when the processor is halted.

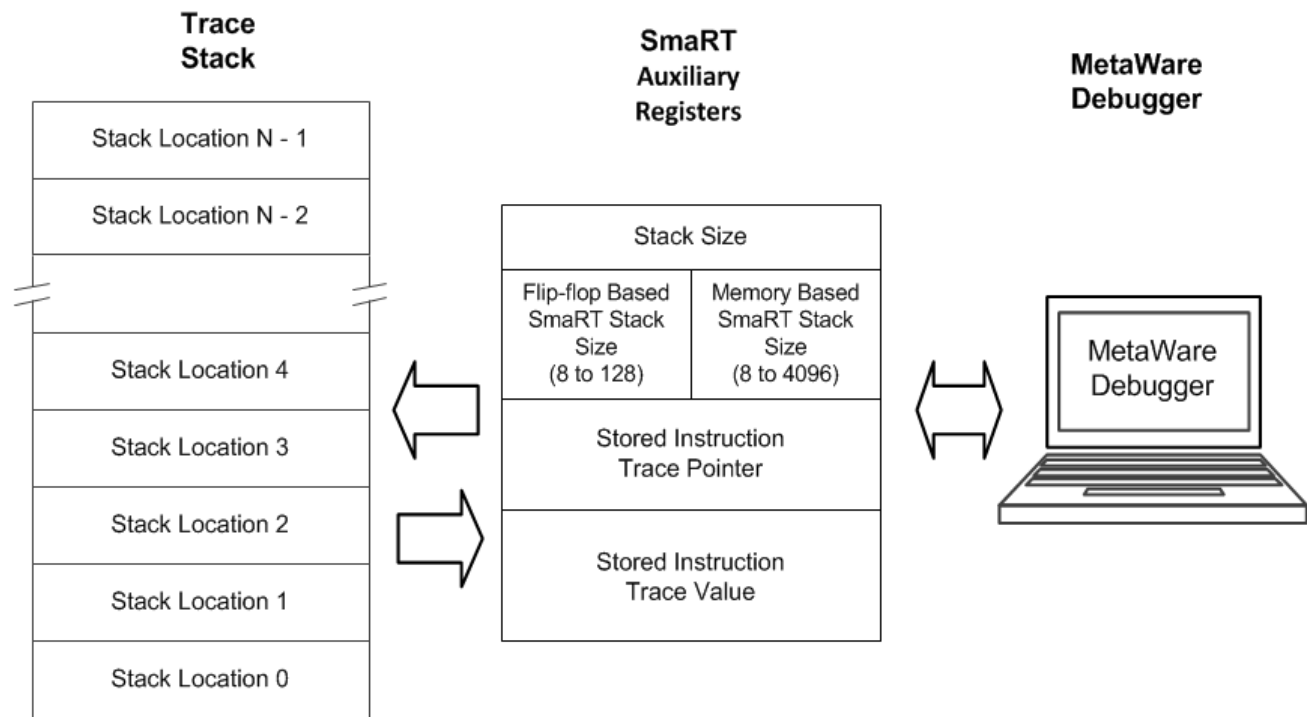
34.3.1 Features

- Different stack sizes available — from 8 up to 4096 entries — to record program-flow changes.
- Any non-sequential instruction that is executed is stored including jumps, branches, interrupts and exceptions, and loops. Multiple branches to the same location (inner loop) are stored as a single entry to maximize stack usage.

- Power-saving features include gating the system clock and zeroing all inputs when not in trace mode.
- The MetaWare debugger enables SmaRT and displays trace information, eliminating the need for a separate, dedicated trace port.

34.3.2 Overview of SmaRT Operation

Figure 34-28 SmaRT Operation



- Trace Stack - stores source and destination addresses whenever a change in program flow is detected by the core. When an interrupt or exception is taken, the address of the vector is not recorded; the target address of the vector, the address of the first instruction of the handler, is recorded.

- SmaRT Auxiliary Registers:
 - SMART_BUILD — shows the configured size of the trace stack
 - SMART_CONTROL — controls SmaRT operation and points to a location in the trace stack
 - SMART_DATA — shows the value of a location in the trace stack (as defined in the SMART_CONTROL register)
- MetaWare Debugger — enables SmaRT and reconstructs the instruction trace

34.3.3 Auxiliary Registers

The SmaRT unit contains the following auxiliary registers:

Register Address	Name	Read/Write	Description
0xFF	SMART_BUILD	R	SmaRT build-configuration register
0x700	SMART_CONTROL	R/W	SmaRT control register
0x701	SMART_DATA	R	SmaRT data register

34.3.4 SMART_BUILD Configuration Register, SMART_BUILD

Address: 0xFF

Access: R

The SMART_BUILD read-only configuration register is at auxiliary address 0xFF and has the following contents:

Figure 34-29 SMART_BUILD Value

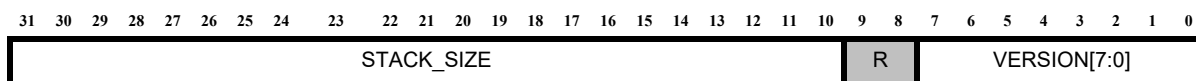


Table 34-11 lists the field descriptions:

Table 34-11 SMART_BUILD

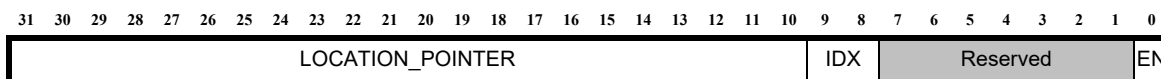
Field	Bits	Description
VERSION	[7:0]	■ 0x3: initial version ■ 0x4: SmARt now retains trace data even after the core is reset. ■ 0x6: Support for Task Switch component.
R	[9:8]	Reserved Returns zero when read.
STACK_SIZE	[31:10]	Trace stack size Indicates the stack size and allows the debugger to identify the capabilities of the implementation for display purposes. Possible values are from 8 up to 4096. Values above 128 are only available for the SRAM-based implementation.

34.3.5 SmaRT Control Register, SMART_CONTROL

Address: 0x700

Access: RW

Figure 34-30 SMART_CONTROL Register



This register lets you enable or disable SmaRT tracing.



Note

This register does not have a reset value. Hence, you must write to this register after reset before the first read attempt.

Table 34-12 lists the field descriptions:

Table 34-12 SMART_CONTROL Register

Field	Bits	Description
EN	[0]	SmaRT enable Enables the trace mechanism. ■ 0x0 : Trace Disabled. ■ 0x1 : Trace Enabled. Upon a hardware (core) reset, the EN bit is reset to 0 (smart trace function is disabled), but all entry data and pointers are retained to enable tracing the events that led to the hardware reset. Resetting smart trace data (entries) and internal entry read/write pointers can only be done using software reset: by toggling the EN bit from 0 to 1.
Reserved	[7:1]	Reserved Returns zero when reading. Ignored when writing.
IDX[1:0]	[9:8]	Location index Controls which value is returned in the SMART_DATA register. ■ 0x0: SRC_ADDR value ■ 0x1: DEST_ADDR value ■ 0x2: FLAGS value ■ 0x3: Reserved

Table 34-12 SMART_CONTROL Register

Field	Bits	Description
LOCATION_POINTER[21:0]	[31:10]	Stack location pointer Sets the value of the stored instruction stack pointer.

**Note**

Reading an entry outside the specified SmaRT stack size returns 0 data.

34.3.8 DEST_ADDR Value, Index 1

Table 34-14 SRC_ADDR Value

Field	Bits	Description
LOCATION_VALUE	[31:0]	Stored instruction trace stack location value The value of the source address in the stack contained at the location given in the LOCATION_POINTER field of the SMART_CONTROL register.

The DEST_ADDR value is read from the SMART_DATA register when IDX field in the SMART_CONTROL register is set to 1.

Figure 34-33 DEST_ADDR Value Register

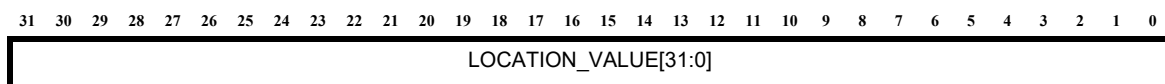


Table 34-15 lists the field descriptions:

Table 34-15 DEST_ADDR Register

Field	Bits	Description
LOCATION_VALUE	[31:0]	Stored instruction-trace stack-location value The value of the destination address in the stack contained at the location given in the LOCATION_POINTER field of the SMART_CONTROL register.

34.3.9 FLAGS Value, Index 2

The FLAGS value is read from the SMART_DATA register when the IDX field in the SMART_CONTROL register is set to 2. The field descriptions are as follows:

Figure 34-34 FLAGS Value Register

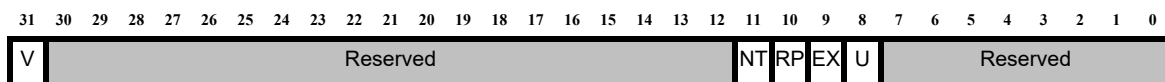


Table 34-16 lists the field descriptions:

Table 34-16 FLAGS Value Register

Field	Bits	Description
Reserved	[7:0]	Reserved Returns zero when read.
U	[8]	User mode Set if the processor is in user mode after program-flow change.

Table 34-16 FLAGS Value Register

Field	Bits	Description
EX	[9]	Exception Indicates that the program-flow change is a result of an exception or interrupt.
RP	[10]	Repeat Used to indicate that the same program-flow change occurred more than once.
NT	[11]	NextTask Indicates that the program flow is changed by the NEXTTASK instruction.
Reserved	[30:10]	Reserved Returns zero when read.
V	[31]	Valid Indicates that the value in the stack is valid.

34.3.9.1 Reserved Value, Index 3

Zero is read from the SMART_DATA register when IDX field in the SMART_CONTROL register is set to 3.

34.4 Debug Interface

When `dbg_en_option` this option is set to true, an external input signal is added to the BVCI interface. When set to high, debug access to the core through the ARC JTAG module is allowed. When cleared all debug accesses through the ARC JTAG module are silently ignored by the debug port and 0 is returned on the read data bus. The core also included an optional APB port using which you can debug the core. You can add the APB port using the `-dbg_apb_option==true`. The APB port has a select input (`dbg_prot_sel`) to control whether this new port or the existing BVCI port is active. Only one port can be active at any time.

The Debug interface is made available for systems that require access to the internals of the core. The interface is composed of memory mapped setup registers that can be written to initiate read and write operations to internal core/auxiliary registers and memory locations, as well as control operations such as instruction-stepping. The debug port is accessed from the JTAG port in the actual design, but may be alternatively connected to a simulation model for verification purposes only (using the `-fast_rascal` option).

This APB port allows access to the debug registers that initiate accesses to the core pipeline, as well as a CoreSight ROM table for device identification. It occupies a single 4 KB block within the Debug Access Port (DAP) address map. For multi-core builds, one 4 KB aperture is used for each core as each core has its own unique APB debug port.

All the internal cores and auxiliary registers of the ARC core, including those for programming any attached ARC Trace module, and any memories accessible from the core can be accessed through this interface. The memory map is the same as that used by the core.

34.4.1 ARC Debug-Access Registers

Table 34-17 on page 1083 lists the names, addresses, and reset values of the ARC debug-access registers.

Table 34-17 Debug Register Addresses

Auxiliary Address	Address over Debug Port	Reset value	Name
0xFFFF_0000	0x0000	0x24	DB_STATUS
0xFFFF_0004	0x0004	0x0	DB_CMD
0xFFFF_0008	0x0008	0x0	DB_ADDR
0xFFFF_000C	0x000C	0x0	DB_DATA
0xFFFF_0010	0x0010	0x0	DB_RESET

34.4.2 Debug Register, DB_STATUS

Address: 0xFFFF_0000

Access: RW

Reset: 0x0000_0000

The DB_STATUS register provides information on all DEBUG operations. Writing to this register initiates the debug operation defined in DB_CMD.

Figure 34-35 DB_STATUS Register

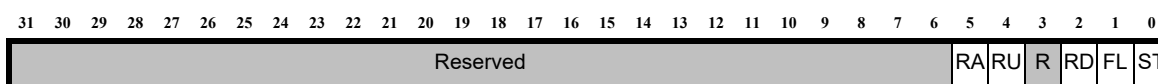


Table 34-18 DB_STATUS Field Description

Field	Bits	Reset	Description
ST	[0]	0	The stalled (ST) flag indicates that the debug interface is busy.
FL	[1]	0	When true, the failure (FL) flag indicates that the debug operation has failed
RD	[2]	0	The ready (RD) flag indicates whether the debug interface is available to accept another transaction command.
RESERVED	[3]	0	Reserved. Read-as-zero; ignored on write
RU	[4]	0	The run (RU) flag is set to one when the ARC EM processor is running
RA	[5]	0	The Reset Applied (RA) flag is used by the debug host to determine that a target reset has occurred.
RESERVED	[31:6]	0	Reserved. Read-as-zero; ignored on write

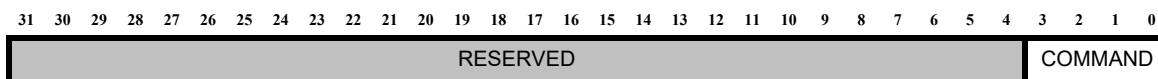
34.4.3 Debug Register, DB_CMD

Address: 0xFFFF_0004

Access: RW

Reset: 0x0000_0003

Figure 34-36 DB_CMD Register



The DB_CMD register defines the debug command (read/write to aux/core/mem).

Table 34-19 DB_CMD Field Description

Field	Bits	Reset	Description
COMMAND	[3:0]	0x3	Defines the debug command (read/write to aux/core/mem). ■ 0x0: Write to Memory Location ■ 0x1: Write to Core Register ■ 0x2: Write to Auxiliary Register ■ 0x3: Initiate a Core/system reset ■ 0x4: Read Memory Location ■ 0x5: Read Core Register ■ 0x6: Read Auxiliary Register ■ 0x7-0xF: Reserved
RESERVED	[31:4]	0	Reserved. Read-as-zero; ignored on write

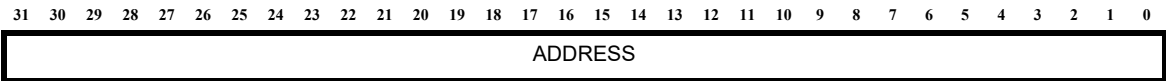
34.4.4 Debug Register, DB_ADDR

Address: 0xFFFF_0008

Access: RW

Reset: 0x0000_0000

Figure 34-37 DB_ADDR Register



The DB_ADDR register specifies auxiliary/core/memory address location.

Table 34-20 DB_ADDR Field Description

Field	Bits	Reset	Description
ADDRESS	[31:0]	0	Auxiliary/core/memory address location to be read from for debug read operations or written to for debug write operations.

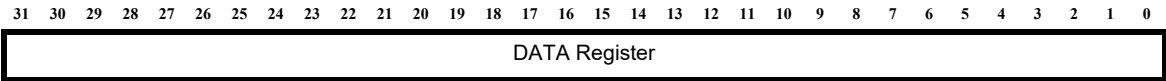
34.4.5 Debug Register, DB_DATA

Address: 0xFFFF_000C

Access: RW

Reset: 0x0000_0000

Figure 34-38 DB_DATA Register



DB_DATA specifies the data to be written for debug write operations. It also holds the returned data for debug read operations.

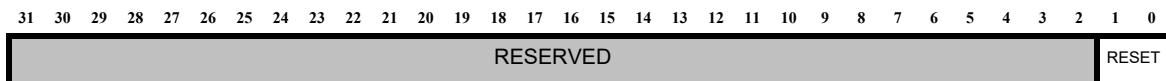
Table 34-21 DB_DATA Field Description

Field	Bits	Reset	Description
DATA_REGISTER	[31:0]	0	Data to be written for debug write operations or returned data for debug read operations.

34.4.6 Debug Register, DB_RESET

Address: 0xFFFF_0010
 Access: Write Only
 Reset: 0x0000_0000

Figure 34-39 DB_RESET Register



Writing any non-zero value to the DB_RESET register triggers a system reset request.



Note

A BVCI write to this register resets the DB_CMD, DB_DATA and DB_ADDR registers.

Table 34-22 DB_ADDR Field Description

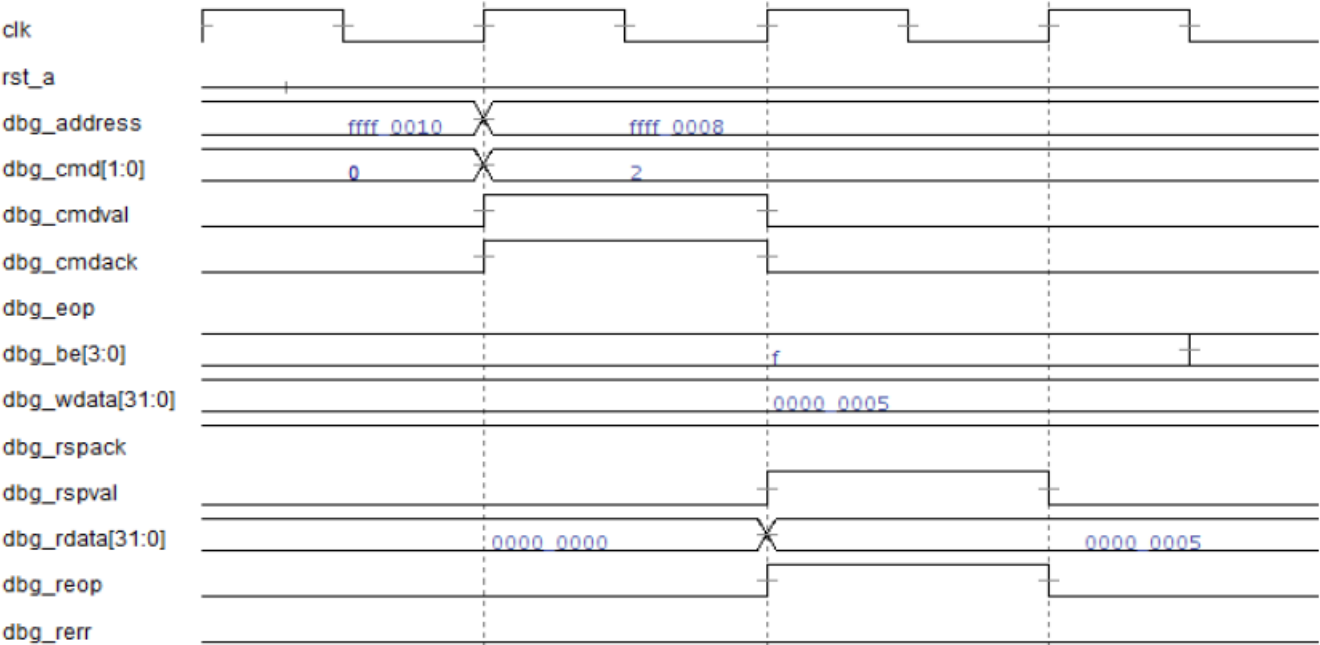
Field	Bits	Reset	Description
RESET	[1:0]	0	Triggers a system reset request. 0x0: No system reset request. 0x1: System reset request. 0x2: System reset request. 0x3: System reset request.
RESERVED	[31:2]	0	Reserved. Read-as-zero; ignored on write.

34.4.7 Initiating a Basic Debug Operation

A debug operation is set up by first writing to DB_ADDR, DB_DATA (for write operations), DB_CMD and finally DB_STATUS. For debug read operations, DB_DATA holds the data read from auxiliary, core register space or memory.

The busy or ready bit in DB_STATUS register should be read to determine whether the operation has completed. The fail bit should then be read to determine whether the operation succeeded or failed.

Figure 34-40 Debug Interface Timing Diagram



35

Performance Counters

35.1 Performance Counters Introduction

The ARCV2-based processors include a performance-monitoring system that enables rapid optimization. This system supports optimization of runtime software and can be used for the following:

- Hot-spot analysis — Locate the functions and instructions consuming the most CPU time. Hardware-assisted statistical code profiling provides visibility of instruction-execution frequencies. The system can also capture microarchitecture detail, reveal the average time taken for each instruction, and allow the root causes of instruction stalls and pipeline flushes to be identified.
- Instruction flow metrics — Identify patterns of behavior common to the whole system. An insight into the behavior of the system as a whole can be gained without requiring knowledge of any individual program block. System-level metrics are calculated from hardware counts of instruction and pipeline events. Example metrics include the following:
 - IPC (instructions per clock)
 - Cache-miss rates
 - Branch-prediction accuracy
 - Average load-use penalty
 - Breakdown of stall causes
 - Instruction mix (LD, ST, branch, and so on)
 - Instruction size (16 or 32 bits)
 - Average function size
 - Average instructions between branches
 - Average stack operations per call
 - Time spent in interrupt or exception handlers

Analysis of metrics can highlight techniques for improvement in the program code, compiler configuration, memory-subsystem design, and CPU configuration.

- **Real-time statistics display** — View system parameters live as the system runs. Metrics collected during a whole program run provide a picture of the system behavior. Graphs of metrics show change in real time as a response to workload or user interaction. Viewing plots of key metrics while the system runs lets you rapidly gain an impression of system behavior and response to workload, which can then be investigated further using more detailed analysis methods.

35.1.1 Performance Counters and Architectural Clock Gating

For runtimes that use architectural clock gating, use the `ACG_CTRL.EP` bit to enable or disable the performance counter clock in the kernel mode by the software or in the debug mode through host access. User mode access to this bit generates a privilege violation exception.

For more information, see [“Architectural Clock Gating Control Register, ACG_CTRL”](#) on page 1038.

35.2 Performance Counters Registers

The following registers provided for interacting with the performance-counter block.

Table 35-1 Performance Auxiliary Registers

Address	Auxiliary Register Name	Description
0x250	Count-Value Registers, PCT_COUNTL	Lower order current-count value
0x251	Count-Value Registers, PCT_COUNTH	Higher order current-count value
0x252	Snapshot-Value Registers, PCT_SNAPL	Lower order snapshot value
0x253	Snapshot-Value Registers, PCT_SNAPH	Higher order snapshot value
0x254	Configuration Register, PCT_CONFIG	Counter configuration
0x255	Control Register, PCT_CONTROL	Control register
0x256	Index-Select Register, PCT_INDEX	Index select register
0x257	Minimum Value Registers, PCT_MINMAXL	Min/Max lower order value
0x258	Maximum Value Registers, PCT_MINMAXH	Min/Max high order value
0x259	Address-Range Registers, PCT_RANGEL	Address range low
0x25A	Address-Range Registers, PCT_RANGEH	Address range high
0x25B	User-Flag Register, PCT_UFLAGS	User flags control register

Conditions Registers

The registers shown in [Table 35-2](#) are provided for interaction with the countable conditions block.

Table 35-2 Countable-Conditions Registers

Address	Name	Description
0xF6	Countable Conditions Build Configuration Register, CC_BUILD	Build-configuration register
0x240	Countable Conditions Index Register, CC_INDEX	Set index to access name of condition
0x241	Countable Conditions Name0 Register, CC_NAME0	Read first four characters of condition name, set by CC_INDEX
0x242	Countable Conditions Name1 Register, CC_NAME1	Read second four characters of condition name, set by CC_INDEX

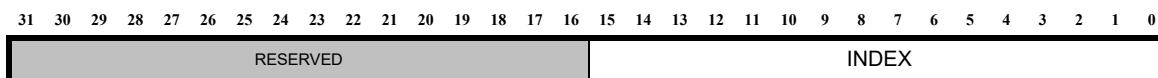
35.2.1 Countable Conditions Index Register, CC_INDEX

Address: 0x240

Access: RW

Reset: 0x0

Figure 35-1 CC_INDEX Register



This register is used to set the index of the countable condition whose name is returned through the CC_NAME0 and CC_NAME1 registers. The conditions that are available is dependent on the processor configuration. For example, if caches are not available in the processor, the cache-related conditions are unavailable. The countable conditions do not have a fixed index. The mapping of named conditions to indexes can be discovered using the CC_INDEX and CC_NAME0 and CC_NAME1 registers.

Table 35-3 CC_INDEX Field Description

Field	Bit	Reset	Description
INDEX	[15:0]	0	Index of the countable condition whose name is returned through the registers CC_NAME0 and CC_NAME1. The index values start at zero.
RESERVED	[31:16]	0	Reserved. Read-as-zero; ignored on write.

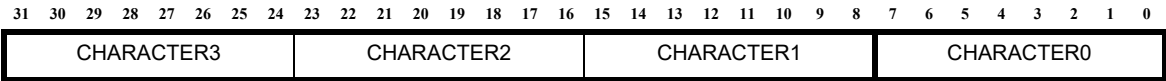
35.2.2 Countable Conditions Name0 Register, CC_NAME0

Address: 0x241

Access: R

Reset: 0x0

Figure 35-2 CC_NAME0 Register



This register provides the eight-character name of the countable condition whose index was written into the CC_INDEX register.

The countable-condition name is stored as a little-endian non-terminated string of ASCII characters. Names are case sensitive. If names are less than eight characters long, trailing zeros are used. Names are unique and allow for identification of each separate condition.

If an invalid index value is set in CC_INDEX, all zeros (null string) is returned.

Table 35-4 CC_NAME0 Field Description

Field	Bit	Reset	Description
CHARACTER0	[7:0]	0	Character 0
CHARACTER1	[15:8]	0	Character 1
CHARACTER2	[23:16]	0	Character 2
CHARACTER3	[31:24]	0	Character 3

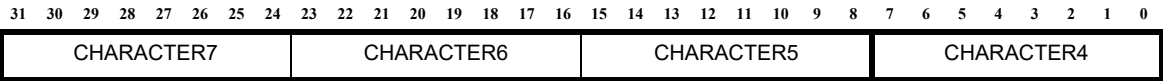
35.2.3 Countable Conditions Name1 Register, CC_NAME1

Address: 0x242

Access: R

Reset: 0x0

Figure 35-3 CC_NAME1 Register



This register provides the eight-character name of the countable condition whose index was written into the CC_INDEX register.

The countable-condition name is stored as a little-endian non-terminated string of ASCII characters. Names are case sensitive. If names are less than eight characters long, trailing zeros are used. Names are unique and allow for identification of each separate condition.

If an invalid index value is set in CC_INDEX, all zeros (null string) is returned.

Table 35-5 CC_NAME1 Field Description

Field	Bit	Reset	Description
CHARACTER4	[7:0]	0	Character 4
CHARACTER5	[15:8]	0	Character 5
CHARACTER6	[23:16]	0	Character 6
CHARACTER7	[31:24]	0	Character 7

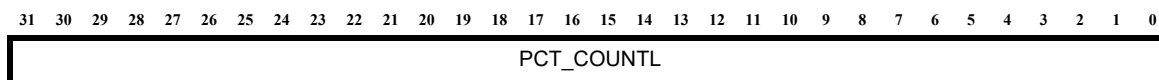
35.2.4 Count-Value Registers, PCT_COUNTL

Address: 0x250

Access: RW

Reset: 0x0

Figure 35-4 PCT_COUNTL Register



Each count value is an unsigned integer, presented in two 32-bit registers. The PCT_INDEX register controls access to the counter:

- The lower-order bits are provided in the PCT_COUNTL register.
- The higher-order bits are provided in the PCT_COUNTH register.

Count values wrap to zero when the maximum count value is reached. If the register is written with a specific value, the counter starts from this value.

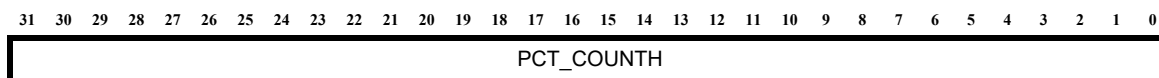
35.2.5 Count-Value Registers, PCT_COUNTH

Address: 0x251

Access: RW

Reset: 0x0

Figure 35-5 PCT_COUNTH Register



Each count value is an unsigned integer, presented in two 32-bit registers. The PCT_INDEX register controls access to the counter:

- The lower-order bits are provided in the PCT_COUNTL register.
- The higher-order bits are provided in the PCT_COUNTH register. If the counter size is 48 bits, the MSB bits of the PCT_COUNTH register [31:16] are read as zero and ignored on writes. If the counter size is 32 bits, the entire PCT_COUNTH register is read as zero and ignored on writes.

Count values wrap to zero when the maximum count value is reached. If the register is written with a specific value, the counter starts from this value.

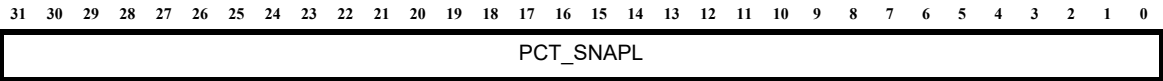
35.2.6 Snapshot-Value Registers, PCT_SNAPL

Address: 0x252

Access: RW

Reset: 0x0

Figure 35-6 PCT_SNAPL Register



The snapshot feature allows simultaneous capture of all count values without halting ongoing counting. Two registers are provided to access the snapshot counter values. Each count value is an unsigned integer, presented in two 32-bit registers. The PCT_INDEX register controls access to the counter snapshot:

- The lower-order bits are provided in the PCT_SNAPL register.
- The higher-order bits are provided in the register PCT_SNAPH register.

Count values from each counter are copied into the associated snapshot registers when a snapshot is triggered through the PCT_CONTROL register. Index selection has no effect on snapshots.

When a counter snapshot is taken, and a counter is in a min or max mode, the contents of the MINMAXL/H registers are copied into the snapshot registers.

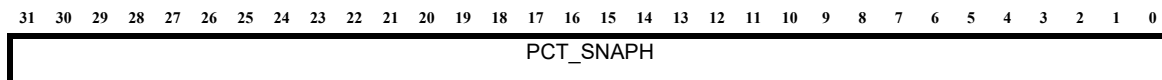
35.2.7 Snapshot-Value Registers, PCT_SNAPH

Address: 0x253

Access: RW

Reset: 0x0

Figure 35-7 PCT_SNAPH Register



The snapshot feature allows simultaneous capture of all count values without halting ongoing counting. Two registers are provided to access the snapshot counter values. Each count value is an unsigned integer, presented in two 32-bit registers. The PCT_INDEX register controls access to the counter snapshot:

- The lower-order bits are provided in the PCT_SNAPL register.
- The higher-order bits are provided in the register PCT_SNAPH register. If the counter size is 48 bits, the MSB bits of the PCT_SNAPH register [31:16] are read as zero and ignored on writes. If the counter size is 32 bits, the entire PCT_SNAPH register is read as zero and ignored on writes.

Count values from each counter are copied into the associated snapshot registers when a snapshot is triggered through the PCT_CONTROL register. Index selection has no effect on snapshots.

When a counter snapshot is taken, and a counter is in a min or max mode, the contents of the MINMAXL/H registers are copied into the snapshot registers.

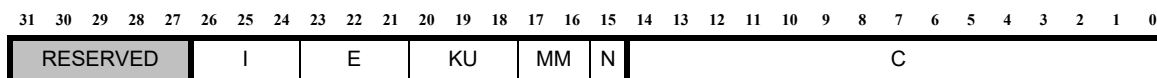
35.2.8 Configuration Register, PCT_CONFIG

Address: 0x254

Access: RW

Reset: 0x0

Figure 35-8 PCT_CONFIG Register



Use this register to configure the counter selected using the PCT_INDEX register.

The condition numbers vary between implementations. Use the countable-conditions registers to determine the allocation of countable-condition names and numbers on the target being accessed.

Table 35-6 PCT_CONFIG Field Description

Field	Bit	Reset	Description
C	[14:0]	0	Countable condition index number
N	[15]	0	Invert condition before use
MM	[17:16]	0	Defines the counting mode 0x0: normal counting 0x1: min counting 0x2: max counting 0x3 to 0x4: reserved
KU	[20:18]	0	0x0: count always 0x1: count only when in user mode 0x2: count only when in kernel mode 0x3 to 0x7: reserved
E	[23:21]	0	0x0: count always 0x1: count only when an exception is active 0x2: count only when an exception is not active 0x3 to 0x7: reserved
I	[26:24]	0	0x0: count always 0x1: count only when an interrupt is taken 0x2: count only when an interrupt is not taken 0x3 to 0x7: reserved

Table 35-6 PCT_CONFIG Field Description

Field	Bit	Reset	Description
RESERVED	[31:27]	0	Reserved. Read-as-zero; ignored on write.

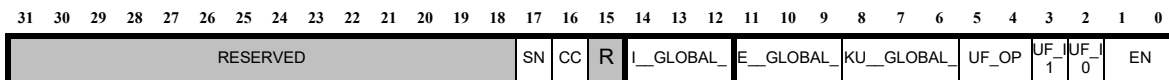
35.2.9 Control Register, PCT_CONTROL

Address: 0x255

Access: RW

Reset: 0x0

Figure 35-9 PCT_CONTROL Register



The global set counter run mode is used to choose all counters enable condition, such as count in kernel mode or user mode, count when an exception is active or inactive, or count when an interrupt is active or inactive. And for each counter, the PCT_CONFIG register controls the local counter enable conditions.

Note that features enabled/disable through programming this register would be effective after the write to this register is completed. For example - ISA events i4byte, i4lbyte, iall, isr, ilr might be triggered based on the instruction format that updates this register. If the instruction enables counting, the events generated by the instruction itself are not counted. If an instruction disables counting, the events are counted.

The counters only count when both local and global counter conditions can be met simultaneously. For example, if the local counter is programmed to count in the kernel mode and when in exception, and the global counter is programmed to count in kernel mode, then the counter counts only in the kernel mode and when in exception. If the global and local counters are programmed such that both the conditions cannot be met at the same time (for example, the local counter is set to count in user mode and the global counter is set to count in the kernel mode), the counter result is zero.

Table 35-7 PCT_CONTROL Field Description

Field	Bit	Reset	Description
EN	[1:0]	0	Enable condition for all counters 0x0: Disable counting 0x1: Enable counting (global enable) 0x2: Count when uflag 0 (user flag 0) and uflag 1 (user flag 1) conditions (as defined in bits [5:2] of this register) are met. 0x3: Global set count run mode (U/K/1, E/NE/1, I/NI/1) This bit has read and write access.
UF_I0	[2]	0	0x0: Use uflag0 as is in the condition UF_OP (bits [5:4]) 0x1: Invert uflag0 (UF_I0) before using in the condition UF_OP (bits [5:4])

Table 35-7 PCT_CONTROL Field Description

Field	Bit	Reset	Description
UF_I1	[3]	0	0x0: Use uflag1 as is in the condition UF_OP (bits [5:4]) 0x1: Invert uflag1 (UF_I1) before using in the condition UF_OP (bits [5:4])
UF_OP	[5:4]	0	User flag operand 0x0: enable counting when (uflag0 XOR UF_I0) AND (uflag1 XOR UF_I1) == 1 0x1: enable counting when (uflag0 XOR UF_I0) OR (uflag1 XOR UF_I1) == 1 0x2: enable counting when (uflag0 XOR UF_I0) == 1 0x3: enable counting when (uflag1 XOR UF_I1) == 1
KU_GLOBAL_	[8:6]	0	0x0: count always 0x1: count only when in user mode 0x2: count only when in kernel mode 0x3-0x7: reserved
E_GLOBAL_	[11:9]	0	0x0: count always 0x1: count only when an exception is active 0x2: count only when an exception is NOT active 0x3-0x7: reserved
I_GLOBAL_	[14:12]	0	0x0: count always 0x1: count only when an interrupt is active 0x2: count only when an interrupt is NOT active 0x3-0x7: reserved
RESERVED	[15]	0	Reserved. Read-as-zero; ignored on write.
CC	[16]	0	Clear counts. When this bit is written with 1, all counters are cleared to zero. Write-only – returns zero on read
SN	[17]	0	Snapshot. When this bit is written with 1, all count values are copied into the snapshot registers. When a counter is in a min/max mode, the min/max register is copied to the snapshot register, instead of the current-count value. Write-only – returns zero on read
RESERVED	[31:18]	0	Reserved. Read-as-zero; ignored on write.

Note that the use of a global count enable condition other than always or never has effects on min/max behavior. See [“Minimum Value Registers, PCT_MINMAXL”](#).

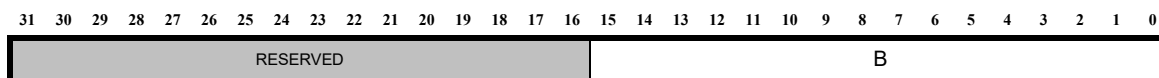
35.2.10 Index-Select Register, PCT_INDEX

Address: 0x256

Access: RW

Reset: 0x0

Figure 35-10 PCT_INDEX Register



This register is used to select the counters to be accessed through count, snapshot, and configuration registers.

Table 35-8 PCT_INDEX Field Description

Field	Bit	Reset	Description
B	[15:0]	0	Counter number. Selects the counters to be accessed through count, snapshot, and configuration registers. Counter operation is not affected by index selection.
RESERVED	[31:16]	0	Reserved. Read-as-zero; ignored on write.

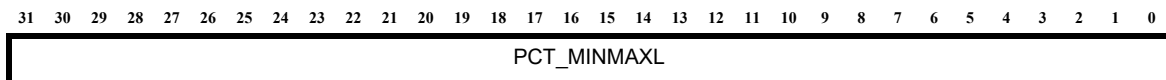
35.2.11 Minimum Value Registers, PCT_MINMAXL

Address: 0x257

Access: RW

Reset: 0x0

Figure 35-11 PCT_MINMAXL Register



Two registers are provided to access the minimum or maximum count value for a continuous sequence of condition-true states:

- The lower-order bits are provided in register PCT_MINMAXL.
- The higher-order bits are provided in register PCT_MINMAXH.

Each count value is an unsigned integer, presented in two 32-bit registers. The min/max value to be accessed is controlled by the PCT_INDEX register.

When a counter snapshot is taken, and a counter is in a min or max mode, the contents of the PCT_MINMAXL and PCT_MINMAXH registers are copied into the snapshot registers.

- Min mode: At the end of an unbroken sequence of condition-true states, the current count value is copied to the min/max register if it is less than the min/max register value.
- Max mode: At the end of an unbroken sequence of condition-true states, the current count value is copied to the min/max register if the count value is greater than or equal to the min/max register value.

Protection logic is included to measure min/max only when the counters are enabled for the entirety of the sequence.

This means the following:

- A sequence is ignored if the condition is true when counters are enabled (by the global-enable register, the global-enable condition, or entry into an address range).
- A sequence is ignored if the condition is true when counters are disabled (by the global-enable register, the global-enable condition, or exit from an address range).

35.2.12 Maximum Value Registers, PCT_MINMAXH

Address: 0x258

Access: RW

Reset: 0x0

Figure 35-12 PCT_MINMAXH Register



Two registers are provided to access the minimum or maximum count value for a continuous sequence of condition-true states:

- The lower-order bits are provided in register PCT_MINMAXL.
- The higher-order bits are provided in register PCT_MINMAXH.

Each count value is an unsigned integer, presented in two 32-bit registers. The min/max value to be accessed is controlled by the PCT_INDEX register.

When a counter snapshot is taken, and a counter is in a min or max mode, the contents of the PCT_MINMAXL and PCT_MINMAXH registers are copied into the snapshot registers.

- Min mode: At the end of an unbroken sequence of condition-true states, the current count value is copied to the min/max register if it is less than the min/max register value.
- Max mode: At the end of an unbroken sequence of condition-true states, the current count value is copied to the min/max register if the count value is greater than or equal to the min/max register value.

Protection logic is included to measure min/max only when the counters are enabled for the entirety of the sequence.

This means the following:

- A sequence is ignored if the condition is true when counters are enabled (by the global-enable register, the global-enable condition, or entry into an address range).
- A sequence is ignored if the condition is true when counters are disabled (by the global-enable register, the global-enable condition, or exit from an address range).

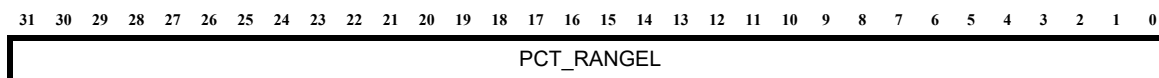
35.2.13 Address-Range Registers, PCT_RANGE1

Address: 0x259

Access: RW

Reset: 0x0

Figure 35-13 PCT_RANGE1 Register



Address range registers can be used to enable counting only when the program counter is within a specific address range. Counting is enabled when $(PCT\_RANGE1 \leq PC)$ and either $(PC < PCT\_RANGEH)$ or $(PCT\_RANGEH == 0)$.

This register defaults to 0x0 on reset.

Note that the use of address ranges has effects on min/max behavior. See [“Minimum Value Registers, PCT_MINMAXL”](#) and [“Maximum Value Registers, PCT_MINMAXH”](#).

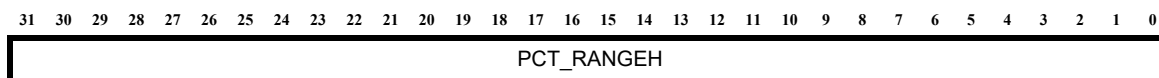
35.2.14 Address-Range Registers, PCT_RANGEH

Address: 0x25A

Access: RW

Reset: 0x0

Figure 35-14 PCT_RANGEH Register



Address range registers can be used to enable counting only when the program counter is within a specific address range. Counting is enabled when $(PCT\_RANGEH \leq PC)$ and either $(PC < PCT\_RANGEH)$ or $(PCT\_RANGEH == 0)$.

Note that PCT_RANGEH is the first instruction outside the range, so to set the high end of the range to the top of memory, set it to 0 (as the first instruction after the top of memory is $0xffffffff+2=0x0$).

This register defaults to 0x0 on reset.

Note that the use of address ranges has effects on min/max behavior. See [“Minimum Value Registers, PCT_MINMAXL”](#) and [“Maximum Value Registers, PCT_MINMAXH”](#).

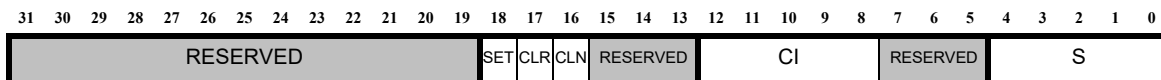
35.2.15 User-Flag Register, PCT_UFLAGS

Address: 0x25B

Access: RW

Reset: 0x0

Figure 35-15 PCT_UFLAGS Register



User-settable flags (uflag x) are provided for instrumenting a specific program. [Table 35-9](#) shows the bit positions for writes.

When the register is read, the current state of the 32-bit flag vector is returned.



Note

You can use the uflag0 and uflag1 (by programming the PCT_CONTROL register) to control the counting of conditions.

The register allows a single aux write to perform both set and clear operations simultaneously.

[Example 35-1](#) describes a few example.

Example 35-1 Example Operations

One-hot operation. Clear all bits, then set flag number N :

```
_sr( (1<<16) | N , AUX_PCT_UFLAGS)
```

Single-bit operation. Clear bit N :

```
_sr( (1<<17) | (N<<8) , AUX_PCT_UFLAGS)
```

Single-bit operation. Set bit N :

```
_sr( (1<<18) | N , AUX_PCT_UFLAGS)
```

Multi-bit operation, Clear bit M , Set bit N :

```
_sr(1<<17) | (M<<8) | (1<<18) | N , AUX_PCT_UFLAGS)
```

Uses

User flags (uflag x) can be used in the following ways:

- Performance counters and ARCPlotter
 - Instrument code into different sections by setting flags, such as uflag0=encode, uflag1=decode, uflag2=protocol, and so on.
 - Counters are programmed to count cycles when uflag0/1 are set.

- Results can be reported at the end of the run, or displayed in real time using the MetaWare debugger's ARCPlotter feature.

**Note**

Only the MetaWare debugger provides live, runtime plotting.

- Performance counters.
 - Instrument code into different sections by setting flags.
 - The counters can be enabled by a condition (or the inverse of a condition).
 - You can then rapidly enable counting for different sections of code by programming the counter hardware to capture data only during `uflag0/uflag1/...` time periods. Different sections can be selected without changing the code.

Table 35-9 PCT_UFLAGS Field Description

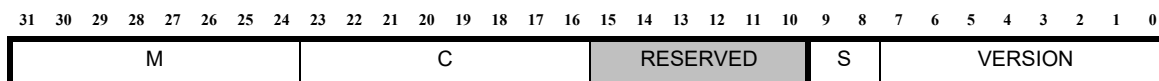
Field	Bit	Reset	Description
S	[4:0]	0	Set-flag index
RESERVED	[7:5]	0	Reserved. Read-as-zero; ignored on write.
CI	[12:8]	0	Clear-flag index
RESERVED	[15:13]	0	Reserved. Read-as-zero; ignored on write.
CLN	[16]	0	Clear all flags
CLR	[17]	0	Clear flag specified by CI field
SET	[18]	0	Set flag specified by the S field, after clearing operations have taken place
RESERVED	[31:19]	0	Reserved. Read-as-zero; ignored on write.

35.2.16 Performance Counter Build-Configuration Register, PCT_BUILD

Address: 0xF5

Access: R

Figure 35-16 PCT_BUILD Register



The performance-counter build-configuration register contains the fields shown in [Table 35-10](#) on page [1112](#).

Table 35-10 PCT_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number. 0x0: not present 0x3: build configuration for ARC HS processors 0x4: build configuration for ARC EM
S	[9:8]	Counter size: 0x0: 32-bit 0x1: 48-bit (default) 0x2: 64-bit All other values – reserved Note: The current implementation supports only 48-bit counters.
RESERVED	[15:10]	Reserved. Read-as-zero; ignored on write.
C	[23:16]	Number of performance counters; 4, 8, 16, and 32 counters are supported. Other values are reserved.
M	[31:24]	Number of counters with min/max capability (starting at counter 0)

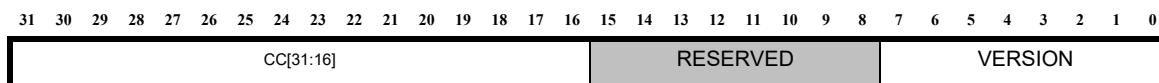
1517

35.2.17 Countable Conditions Build Configuration Register, CC_BUILD

Address: 0xF6

Access: R

Figure 35-17 CC_BUILD Register



This register indicates the following:

- Presence of the countable-conditions block in the system
- Version number of the block
- Number of countable conditions available.



Note

The version number may change in future, allowing the programming interface through the other CC_* registers to be altered.

Countable conditions may be added or removed from the list without changing the version number. Software should always interrogate the hardware to determine which conditions are available on the target being used.

Table 35-11 CC_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number. All non-enumerated values are reserved. ■ 0x0 if no countable conditions are present ■ 0x4 for countable conditions
CC	[31:16]	The number of countable conditions present in the system

36

ARC Trace

36.1 ARC Trace Introduction

ARC Trace is a real-time state tracing system for ARCV2-based processors. ARC Trace allows you to trace the following state information for a processor:

- Data collection: data read and write to auxiliary registers, core register writes, and memory read and writes.
- Debug trace: Program flow, process ID trace, and watchpoint trace.

You can use the MetaWare debugger to enable ARC Trace and display the debug trace.

36.2 ARC Trace Auxiliary Registers

Table 36-1 ARC Trace Registers

Address	Auxiliary Register Name	Description
0xF2	ARC Trace Build Configuration Register, RTT_BUILD	Build configuration register
0x380	ARC Trace Address Register, RTT_ADDRESS	Contains the ARC Trace memory address that you want to access
0x381	ARC Trace DATA Register, RTT_DATA	Contains the data that you want to read or write to ARC Trace
0x382	ARC Trace CMD Register, RTT_COMMAND	ARC Trace command register

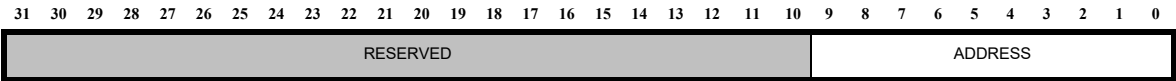
36.2.1 ARC Trace Address Register, RTT_ADDRESS

Address: 0x380

Access: RW

Reset: 0x0000_0000

Figure 36-1 RTT_ADDRESS Register



This register specifies the ARC Trace memory address which you want to write to or read from. The address written to this register is presented on the ARC Trace interface address bus; the desired address must be set up prior to writing the RTT_COMMAND register, which initiates the ARC Trace read or write operation.

The operation that the processor performs on this ARC Trace memory address is specified by the RTT_COMMAND register.

The RTT_DATA register contains the data read from this ARC Trace memory address or the data you want to write to this memory address.

For more information about this register, see the *ARC Trace Databook*.

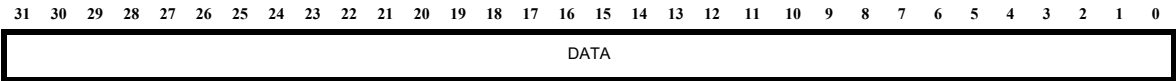
36.2.2 ARC Trace DATA Register, RTT_DATA

Address: 0x381

Access: RW

Reset: 0x0000_0000

Figure 36-2 RTT_DATA Register



This register contains the data read from the ARC Trace or the data to be written to the ARC Trace. The read or write operation is determined by the value of the RTT_COMMAND register. The RTT_ADDRESS register specifies the ARC Trace memory address that is being accessed. For more information about this register, see the *ARC Trace Databook*.

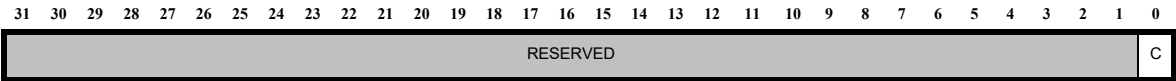
36.2.3 ARC Trace CMD Register, RTT_COMMAND

Address: 0x382

Access: W

Reset: 0x0000_0000

Figure 36-3 RTT_COMMAND Register



This register specifies the actions that the processor performs on ARC Trace.

Table 36-2 RTT_COMMAND Bit Field Description

Field	Bit	Description
C	[0]	Specifies the action performed on ARC Trace 0: indicates a read operation from ARC Trace 1: indicates a write operation to ARC Trace

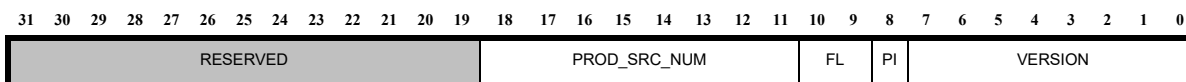
For more information about this register, see the *ARC Trace Databook*.

36.2.4 ARC Trace Build Configuration Register, RTT_BUILD

Address: 0xF2

Access: R

Figure 36-4 RTT_BUILD Register



This register specifies the build-time configuration of ARC Trace.

Table 36-3 RTT_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of ARC Trace ■ 0x1: Initial version of ARC Trace ■ 0x2: Second version of ARC Trace ■ 0x3: Third version of ARC Trace ■ 0x4: Fourth version: - Added support for CoreSight™ - Modified trace buffers in CoreSight builds to use flip flop arrays not RAMs. ■ 0x5: Fifth version. Indicates support for both nexus interface and CoreSight. However, you can configure only one interface in a processor build. ■ 0x6: Includes support for: - NEXTTASK instruction based jump - RTT_PRDCR_BLD_REG register that indicates if the producer supports the NEXTTASK instruction
PI	[8]	Indicates the presence of ARC Trace programming interface This bit is always set to 1; it means that you can program the ARC Trace control registers using this core.
FL	[10:9]	Indicates the feature level of the producer ■ 0x0: small producer; defines program counter as a source for trace messages ■ 0x1: medium producer; defines program counter and memory read/write as sources for trace messages ■ Other values are: reserved

Table 36-3 RTT_BUILD Field Descriptions (Continued)

Field	Bit	Description
PROD_SRC_NUM	[18:11]	For an ARC EM22FS processor operating in the performance mode, this field returns 0 for the main core and 1 for the safety core. For other ARC EM processors, this field returns 0.
RESERVED	[31:21]	Reserved. Read-as-zero; ignored on write.

For more information about this register, see the *ARC Trace Databook*.

36.3 ARC Trace Address Map

ARCV2 ISA has the following kinds of registers:

- [ARC Trace Control Registers](#)
- [ARC Trace Producer Build Configuration Register, RTT_PRDCR_BLD_REG](#)
- [ARC Trace Producer Registers](#)

ARC Trace Address Map is shown in [Table 36-4](#).

Table 36-4 ARC Trace Address Map

Bank	Start Address	End Address
Control	0x000	0x01F
Config (Nexus Offload)	0x020	0x03F
ARC Producer (Nexus Offload)	0x040	0x07F

36.4 ARC Trace Control Registers

You can use the ARC Trace control registers to program the ARC Trace at set-up time. These registers are not common to all producers. Each producer has an independent set of control registers.

[Table 36-5](#) summarizes the build configuration registers, control registers, and status registers for components that are described in this document.

Table 36-5 ARC Trace Control Registers

Offset from Base Address (0x00)	Name	Registers Present in the Configurations
0x00	ARC Trace I/F Build Configuration Register, ARCT_BUILD	All
0x01	Trace Producer Select Register, ARCT_PRSEL	All
0x04	Trace Timestamp Enable Register, ARCT_TSTAMP_ENABLE	All
0x05	Trace Debugger Message Enable Register, ARCT_DEBUGGER_MESSAGE_ENABLE	All
0x10	ARC Trace Producer Build Configuration Register, RTT_PRDCR_BLD_REG	All

36.4.1 ARC Trace I/F Build Configuration Register, ARCT_BUILD

Offset from Base 0x00
 Address:
 Access: R
 Reset Build-time configuration values from
 ARChitect

Figure 36-5 ARCT_BUILD Register

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
Reserved					SWEI D_W	OCM_W	F	GS	CS	NIF	MSEO _W	MDO_W	OCM	INT_MEM_SIZE	NUM_PROD					VERSION											

This register specifies the build-time configuration of ARC Trace. This information is presented as an aid to connected debuggers to quickly identify the trace resources and message content, format, stream interface.

Table 36-6 ARCT_BUILD Field Descriptions

Field	Bits	Description
VERSION	[7:0]	Version of ARC Trace 0x05 = Includes the following: - Includes support for both Nexus interface and CoreSight offload interface. - Added support for the system timestamp message for nexus builds - Relocated the on-chip memory-related registers to the configuration register address space (these registers are common to all producers) - Added program control when to generate resource full message for conditional branches - Added program control to include cache miss and cache hit information in a data trace message. 0x6: Includes support for: - NEXTTASK instruction based jump - RTT_PRDCR_BLD_REG register that indicates if the producer supports the NEXTTASK instruction
NUM_PROD	[11:6]	Number of trace producers. For an ARC EM22FS processor operating in the performance mode, this field reads as 2. Otherwise, this field reads as 1.
INT_MEM_SIZE	[14:12]	Internal trace memory size. All non-enumerated values are reserved. 0x0 : 4K 0x1 : 8K 0x2 : 6K 0x3 : 32K 0x4 : 64K 0x5 : 128K

Table 36-6 ARCT_BUILD Field Descriptions (Continued)

Field	Bits	Description
OCM	[15]	On-Chip Memory Support 0x1: On-Chip Memory Present 0x0: On-Chip Memory Not Present For CoreSight builds. This bit always reads 0.
MDO_W	[17:16]	Number of Nexus ports present in the build. All non-enumerated values are reserved. 0x0 : Single Nexus port 0x1: Dual Nexus ports
MSEO_W	[18]	Nexus MSEO Width This bit is always 1. This indicates that a 2-bit control signal (MSEO) is used.
NIF	[19]	Nexus interface present 0x1: Nexus Interface present 0x0: Nexus Interface not present For CoreSight builds. This bit always reads 0.
CS	[20]	CoreSight compatible (ATB) interface present. 0x1: Coresight Interface present 0x0: Coresight Interface not present
GS	[21]	Graduating store feature present (HS 4xD only). The data trace write message is affected if this feature is configured. 0x1: Graduating Store present 0x0: Graduating Store not present
F	[23:22]	Address Filter Pairs 0x0: 8 0x1: 4 0x2: 2 0x3: 0
OCM_W	[25:24]	On-Chip Memory width 0x0 : 32 bit 0x1: 64 bit 0x2: 128 bit For CoreSight builds. this bit always reads 00.
SWEID_W	[26]	Width of Software Event IDTAG 0x0: (4bits) Other values are reserved
RESERVED	[31:27]	Reserved. Read-as-zero; ignored on write.

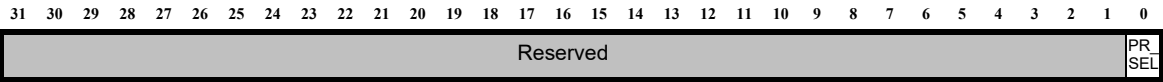
36.4.2 Trace Producer Select Register, ARCT_PRSEL

Offset from Base Address: 0x01

Access: RW

Reset: 0x0

Figure 36-6 ARCT_PRSEL Register



Enables trace collection for this producer.

Table 36-7 ARCT_PRSEL Field Descriptions

Field	Bits	Reset	Description
PR_SEL	0	0	Producer Select 0x1: Producer is active 0x0: Producer is not active
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

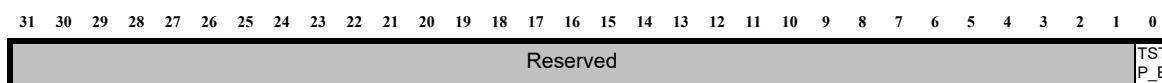
36.4.3 Trace Timestamp Enable Register, ARCT_TSTAMP_ENABLE

Offset from base address: 0x04

Access: RW

Reset: 0x0

Figure 36-7 ARCT_TSTAMP_ENABLE Register



Trace messages from this producer may include the optional nexus timestamp field with each message.

Table 36-8 ARCT_TSTAMP_ENABLE Field Descriptions

Field	Bits	Reset	Description
TSTP_E	0	0	Producer Timestamp Enable 0x0: Timestamp field is not included in trace messages. 0x1: Timestamp is included in trace messages.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

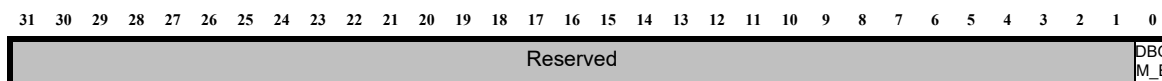
36.4.4 Trace Debugger Message Enable Register, ARCT_DEBUGGER_MESSAGE_ENABLE

Offset from base address: 0x05

Access: RW

Reset: 0x0

Figure 36-8 ARCT_DEBUGGER_MESSAGE_ENABLE Register



Debug Message enable. Enables you to trace the debugger-inserted instructions.

Table 36-9 ARCT_DEBUGGER_MESSAGE_ENABLE Field Descriptions

Field	Bits	Reset	Description
DBG M_E	0	0	Enable ARC Trace to trace debugger-inserted instructions 0x0: Debugger transactions are not included in trace stream for this producer. 0x1: Debugger transactions are included in trace stream for this producer.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

36.4.5 ARC Trace Producer Build Configuration Register, RTT_PRDCR_BLD_REG

Offset from Base 0x10

Address:

Access: R

Figure 36-9 RTT_PRDCR_BLD_REG Register

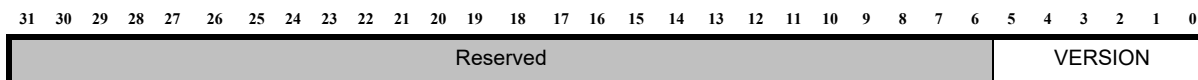


Table 36-10 RTT_PRDCR_BLD_REG Description

Field	Bits	Description
VERSION	[5:0]	Version of Trace Producer 0x0 = No additional feature is implemented 0x1 = NEXTTASK instruction based jump is supported
RESERVED	[31:6]	Reserved. Read-as-zero; ignored on write.

36.5 ARC Trace Configuration Registers

The configuration registers allow the capture probe to manipulate ARC Trace. Only one register is programmed at a time and the Status Read only registers hold the status of the ARC Trace. The configuration registers are available only when Nexus interface or On-Chip Memory is configured. These registers are shared by all producers.



Note

These registers can be set up only before building the configuration, and not while ARC Trace is running.

Table 36-11 lists ARC Trace Configuration Registers

Table 36-11 ARC Trace Configuration Registers

Address	Name	Registers Present in the Configurations
0x30	Trace On-Chip Memory Base Address Register, ARCT_OCM_BASE	With on-chip memory
0x31	Trace On-Chip Memory Size Register, ARCT_OCM_SIZE	
0x32	Trace On-Chip Memory Write Pointer Register, ARCT_OCM_WPTR	
0x34	Trace Transport Status Register, ARCT_TR_STATUS	

Address	Name	Registers Present in the Configurations
0x36	Trace Nexus Offload Control Register, ARCT_FLUSH_CTRL	All
0x37	Trace Nexus Offload Control Register, ARCT_OFFLOAD_CTRL	With on-chip memory or Nexus interface
0x38	Pattern Generation Register, PTRN_GEN	With Nexus offload
0x39	Nexus Clock Control Register, NEXUS_CLK_DIV	
0x1A	EVTI Control Register, PR_EVTI_REG	

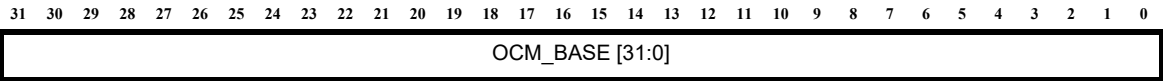
36.5.1 Trace On-Chip Memory Base Address Register, ARCT_OCM_BASE

Address: 0x30

Access: RW

Reset: ARChitect configured value

Figure 36-10 ARCT_OCM_BASE Register



ARCT_OCM_BASE holds the absolute address value for the on-chip memory where the AHB/ AXI master stores the Nexus messages from trace. The address must be a 64-byte aligned address for 32-bit data busses and 128-byte aligned for 64-bit data busses.

 **Note** If ADDR_SIZE is less than 32 bits, when programming this register, set the MSB [32:ADDR_SIZE] bits to 0.

Table 36-12 ARCT_OCM_BASE Field Descriptions

Field	Bits	Description
OCM_BASE	[31:0]	Base Address

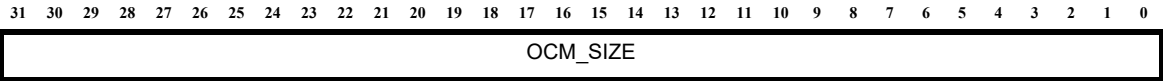
36.5.2 Trace On-Chip Memory Size Register, ARCT_OCM_SIZE

Address: 0x31

Access: RW

Reset: 0x8000 0000

Figure 36-11 ARCT_OCM_SIZE Register



ARCT_OCM_SIZE holds the size of the maximum addressable memory reserved for Trace On-Chip memory where Nexus messages are stored. The value of OCM_SIZE must be in multiples of ARCT_BUILD[Int_Mem_Size].

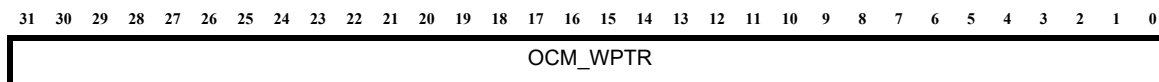
Table 36-13 ARCT_OCM_SIZE Field Descriptions

Field	Bits	Reset	Description
OCM_SIZE	[31:0]	0x8000 0000	Memory size

36.5.3 Trace On-Chip Memory Write Pointer Register, ARCT_OCM_WPTR

Address: 0x32
Access: RW
Reset: 0x0

Figure 36-12 ARCT_OCM_WPTR Register



ARCT_OCM_WPTR holds the write pointer value for On Chip Memory allocated for Trace. This register is updated when a write is issued to OCM. This register holds the relative address and the absolute location of OCM is calculated by adding this write-pointer value to OCM Base Address Register value.

The write pointer is a byte counter and its resolution is dependent on the memory-bus width of the producer. If the producer uses a 64-bit memory bus width, the write pointer is an eight-byte counter. If the producer uses a 32-bit memory bus width, the write pointer is a four-byte counter. When trace messages end in a non-aligned 8-byte location, ARC Trace appends zeros to the MSB bytes.

When wrap is disabled and OCM is full (`TR_STATUS[6] == 1`), ARC Trace is halted and it cannot write to the on-chip memory. To un halt ARC Trace, the write pointer is reset to point to the on chip memory base address. Before un halting ARC Trace, ensure that you have read the on-chip memory to avoid trace data loss.

When wrap is enabled and wrapping occurs, trace data is overwritten.



Note

If ADDR_SIZE is less than 32 bits, when programming this register, set the MSB [32:ADDR_SIZE] bits to 0.

Table 36-14 ARCT_OCM_WPTR Field Descriptions

Field	Bits	Reset	Description
OCM_WPTR	[31:0]	0x0000 0000	Current relative address of OCM write pointer.

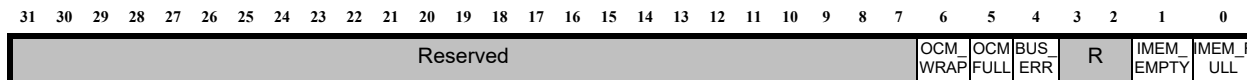
36.5.4 Trace Transport Status Register, ARCT_TR_STATUS

Address: 0x34

Access: R

Reset: 0x0

Figure 36-13 ARCT_TR_STATUS Register



Status of the Trace intermediate buffers and On-Chip Memory storage.

Table 36-15 ARCT_TR_STATUS Field Descriptions

Field	Bits	Reset	Description
IMEM_FULL	[0]	0	Internal Memory Full. 0x1: Full 0x0: Not full This field is valid for both on-chip memory offload and nexus offload configurations.
IMEM_EMPTY	[1]	0	Internal Memory Empty 0x1: Empty 0x0: Not empty This field is valid for both on-chip memory offload and nexus offload configurations.
RESERVED	[3:2]	0	Reserved. Read-as-zero; ignored on write.
BUS_ERR	[4]	0	Internal Memory Bus Error Detected. 0x1: Bus error detected. Bit is cleared when host reads Transport status register 0x0: No Bus error detected. This bit is cleared when the host (debugger) reads the ARCT_TR_STATUS register. This bit is valid for an ARC Trace configuration that includes an on-chip memory offload interface. For other ARC Trace configurations without the on-chip memory interface, this bit is reserved.
OCM_FULL	[5]	0	On-Chip Memory Full. 0x1: Full. 0x0: Not full

Table 36-15 ARCT_TR_STATUS Field Descriptions (Continued)

Field	Bits	Reset	Description
OCM_WRAP	[6]	0	On-chip memory wrap status ■ 0x1: Write pointer has wrapped. ARC Trace inserts '1' in the MSB to each nexus message to make them word-aligned. ■ 0x0: Write pointer has not wrapped. The nexus messages in this case are not word aligned. ARC Trace pads '0' in the MSB after the last nexus message.
RESERVED	[31:7]	0	Reserved. Read-as-zero; ignored on write.

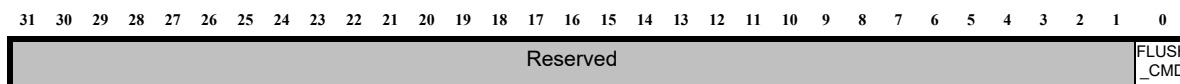
36.5.5 Trace Nexus Offload Control Register, ARCT_FLUSH_CTRL

Address: 0x36

Access: RW

Reset: 0x0

Figure 36-14 ARCT_FLUSH_CTRL Register



This register is used to flush internal buffers and places enqueued messages onto the trace offload interface. Initiate the flush operation by writing a 1 to FLUSH_CMD. This bit is automatically cleared to 0 when all stored messages have egressed.

Table 36-16 ARCT_FLUSH_CTRL Field Descriptions

Field	Bits	Reset	Description
FLUSH_CMD	[0]	0	Flush command bit for Nexus Aux/OCM offload. 0x0: Flush command completed. Writing 0 has no effect. 0x1: Assert 1 to initiate Flush operation.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

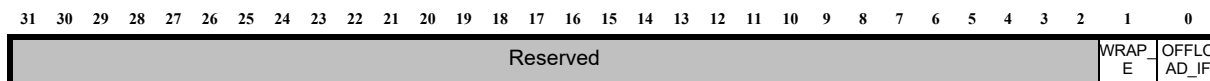
36.5.6 Trace Nexus Offload Control Register, ARCT_OFFLOAD_CTRL

Address: 0x37

Access: RW

Reset: 0x01

Figure 36-15 ARCT_OFFLOAD_CTRL Register



Selects the mode of storage for Nexus trace messages. The Nexus I/F and/or On-Chip Memory must be included in the build to be applicable. You can also enable the wrap feature for On-Chip memory. The wrap feature is a rolling storage buffer that replaces old data with incoming data. If wrap mode is not enabled, an internal memory full indicator is set and new data is lost when the On-Chip Memory is full.

Table 36-17 ARCT_OFFLOAD_CTRL Field Descriptions

Field	Bits	Reset	Description
OFFLOAD_IF	[0]	1	Nexus interface Selection. 0x0: Nexus interface selected for trace egress 0x1: On-chip memory interface is selected for trace storage
WRAP_E	[1]	0	OCM Wrap mode enable. 0x0: Internal memory wrap mode is disabled. ARC Trace stops offloading messages to the on chip memory until the debugger resets the write pointer to the OCM base address 0x1: Internal memory wrap mode is enabled.
Reserved	[31:2]	0	Reserved. Read-as-zero; ignored on write.

When the on-chip memory supports wrapping, the message formats are as follows:

```
32bit : {NTA_ID[3:2], MSEO[1:0], MDO[7:0], MSEO[1:0], MDO[7:0], MSEO[1:0], MDO[7:0]}
        {NTA_ID[1:0], MSEO[1:0], MDO[7:0], MSEO[1:0], MDO[7:0], MSEO[1:0], MDO[7:0]}
```

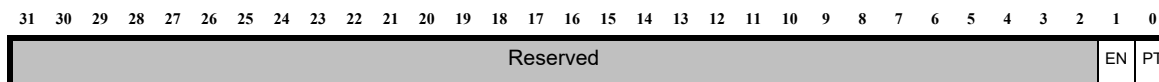
36.5.7 Pattern Generation Register, PTRN_GEN

Address: 0x38

Access: RW

Reset: 0x0

Figure 36-16 PTRN_GEN Register



This register lets you generate a bit pattern on the offload Nexus interface. You can use this bit pattern to tune your debugger to the MCKO output clock of ARC Trace. This register exists only if you have included an offload Nexus interface during the build-time configuration.

Table 36-18 PTRN_GEN Field Descriptions

Field	Bits	Reset	Description
EN	[0]	0	Enable pattern generation on Nexus Interface 0x1: pattern generation is enabled. 0x0: pattern generation is disabled.
PT	[1]	0	Generate specific bit pattern on the Nexus Interface. For eight-bit MDO Nexus interface 0x1: generate the following pattern: 0x55-> 0xAA-> 0x55-> 0xAA 0x0: generate the following pattern: 0x00 -> 0xFF -> 0x00 -> 0xFF
Reserved	[31:2]	0	Reserved. Read-as-zero; ignored on write.

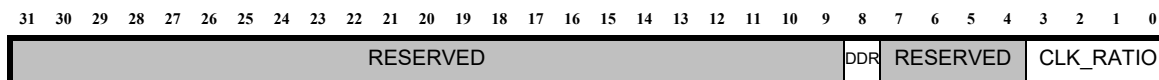
36.5.8 Nexus Clock Control Register, NEXUS_CLK_DIV

Address: 0x39

Access: RW

Reset: 0x01

Figure 36-17 NEXUS_CLK_DIV Register



This register lets you configure the ratio between the system clock and the MCKO output clock of ARC Trace. System clock and ARC Trace clock should be phase aligned to support this clock ratios feature. This register is present only if ARC Trace is built with a Nexus interface.

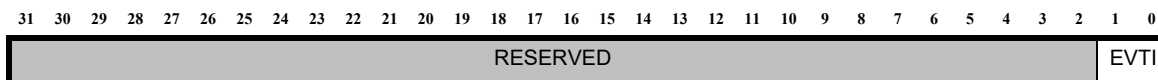
Table 36-19 NEXUS_CLK_DIV Field Descriptions

Field	Bits	Reset	Description
CLK_RATIO	[3:0]	1	Specifies the clock ratio between the system clock and the MCKO output clock The allowed values are: 1,2,3,4,5,6,8 ■ 0x1: enable for clock ratio of 1 (default) ■ 0x2: enable for clock ratio of 2 ■ 0x3: enable for clock ratio of 3 ■ 0x4: enable for clock ratio of 4 ■ 0x5: enable for clock ratio of 5 ■ 0x6: enable for clock ratio of 6 ■ 0x7: enable for clock ratio of 8
Reserved	[7:4]	0	Reserved. Read-as-zero; ignored on write.
DDR (Double-Data Rate)	[8]	0	Specifies if ARC Trace must offload trace messages at both the positive edge and negative edge of the Nexus clock ■ 0x0: indicates that ARC Trace offloads trace messages only at the negative edge of the clock. ■ 0x1: indicates that ARC Trace offloads trace messages at both positive and negative edges of the clock. Note: When the CLK_RATIO == 1, ARC Trace offloads trace messages only on the negative edge of each clock cycle irrespective of the value of the Double-Data rate field.
Reserved	[31:9]	0	Reserved. Read-as-zero; ignored on write.

36.5.9 EVTI Control Register, PR_EVTI_REG

Address: 0x3A
 Access: RW
 Reset: 0x00

Figure 36-18 PR_EVTI_REG Register



This register lets you control ARC Trace behavior when an EVTI signal is asserted from the off-chip Nexus interface. This register is present only if ARC Trace is built with a Nexus interface.

Table 36-20 PR_EVTI_REG Field Descriptions

Field	Bits	Reset	Description
EVTI	[1:0]	0	Enable or disable synchronization messages when an EVTI signal is asserted. 0x0: Synchronize trace messages. The first message of each message type after the EVTI signal assertion is a sync message. For example, if ARC Trace is set to send sync messages on an EVTI signal, the first message after the EVTI assertion of the following trace messages will be a sync message: program trace, data trace read/write, program trace indirect branch history, auxiliary register read/write trace 0x2: Disabled. The EVTI signal is ignored. Other values of this field are reserved.
RESERVED	[31:2]	0	Reserved. Read as zero. Ignore on writes

36.6 ARC Trace Producer Registers

Each producer has instances of the following ARC Trace producer registers. The number of producer register banks is equal to the number of producers configured during build-time in ARChitect.

The Producer Registers allows the capturing probe JTAG interface to manipulate the producer configurations. Only one register can be programmed at a time and the Status Read only registers hold the status of the ARC Trace internal Modules.

[Table 36-21](#) lists the ARC core producer registers.

Table 36-21 ARC Core Producer Registers

Address Offset from Producer Base	Register Name	Registers Present in the Configurations
See Table 36-22	“Address Filter Registers”	

Table 36-21 ARC Core Producer Registers

Address Offset from Producer Base	Register Name	Registers Present in the Configurations
0x14	"Trace Producer Type Register, ARCT_PRTYPE"	All
0x15	"Trace Filter Type Register, ARCT_FLT_TYPE"	All
0x16	"Trace Filter Control Register, ARCT_FLT_CTRL"	All
0x17	"Trace Producer Source Enable Register, ARCT_PRSRC_E"	All
0x18	"Trace Producer Watchpoint Status Register, ARCT_PRWP_STATUS"	All
0x19	"Trace Producer Watchpoint Status Register, ARCT_PRWP_STATUS"	All
0x1A	"Trace Producer Watchpoint Enable Register, ARCT_PRWP_E"	All
0x1E	"Producer Trace Attribute Register, ARCT_PRATTR"	All
0x1F	"Producer CoreSight ID Register, ATID_REG_ADDR"	with CoreSight ATB

36.6.1 Register Description

ARC Trace supports programmable eight address filters and two data filters. Each address filter can be programmed as a global filter, a range filter, or a trigger filter.

A range filter captures the values between the start address and the stop address including the start and stop addresses of a producer's source.

A global filter is a trace-all filter of PC range. Global filter captures the messages of all sources which fall within the global filter PC start and PC stop range.

A trigger filter captures the values from the matched start register address to stop register address including the interrupts and exceptions which are triggered in the range. The trigger register needs to be programmed to select the operation of the filters between range filter and trigger filter.

To select the filter, each range/trigger filter requires a pair of registers to hold the start address and stop address of the source. ARC Trace has eight filter start address registers and eight filter stop registers named as Address Start Filter0 – Address Start Filter7 and Address Stop Filter0 – Address Stop Filter7. Multiple filters can be programmed to the same source. Assigning a range filter to a producer source is done by programming the filter type register.

When a global filter is programmed as a trigger filter, it starts capturing the trace from the start address of the global trigger filter and stops when it matches to stop address global trigger.

If both global trigger and a source trigger are active, source trigger is active only in global range. The source trigger is disabled when the address matches to the stop address of the source or it is out of the global filter range. The programming and its use cases are explained in Global filters section.

ARC Trace data filters named as Data Filter0 and Data Filter1 hold the data of the source to be matched. ARC Trace generates the Nexus message when trace data matches with the filter data. The data filters support both 32-bit and 64-bit data width. If data width is 64-bit, the data filters have two 32-bit registers to hold the 64-bit data ie., Data Filter(0/1) LSW and Data Filter(0/1) MSW, else only data filter(0/1) LSW exist.

36.6.2 Address Filter Registers

ARC Trace supports up to eight address filters. The number of filters included in the build, is configurable at built time. Use the `-rtt_num_filters` option to configure the number of filters.

The address filters can be programmed to capture the data of the following sources:

- Program counter
- Memory Write Data
- Memory Read Data

Data filters support all the sources except Program Counter.

When the filter registers are programmed for a source, the programming filters called with the source name as listed below:

- Program counter Filter start register
- Program counter Filter stop register
- Memory Write Filter start register
- Memory Write Filter stop register
- Memory Read Filter start register
- Memory Read Filter stop register

Each address filter has a start and stop address register associated with it. [Table 36-22](#) shows the start address and stop address filters.

Table 36-22 Start Address Register and Stop Register Address Offsets

Filter	Start Address Register Offset	Stop Address Register Offset
Filter0	0x000	0x001
Filter1	0x002	0x003
Filter2	0x004	0x005
Filter3	0x006	0x007
Filter4	0x008	0x009
Filter5	0x00A	0x00B

Table 36-22 Start Address Register and Stop Register Address Offsets

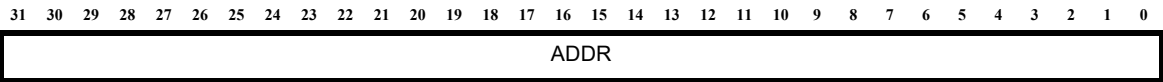
Filter	Start Address Register Offset	Stop Address Register Offset
Filter6	0x00C	0x00D
Filter7	0x00E	0x00F

36.6.2.1 **Start Address Register, ARCT_ADSTR_FLTx**

Access: RW

Reset: 0x00

Figure 36-19 Start Address Register



The PC start address register holds the value of the starting address of the PC for the range filter. A Range filter is a filter captures the values between the start value and stop value of a producer's source including the boundaries. The Range filter starts capturing the trace soon after the PC address value is matched or greater than the PC filter start address.

 **Note** If PC_SIZE is less than 32 bits, when programming this register, set the MSB [32:PC] bits to 0.

Table 36-23 Start Address Register Field Descriptions

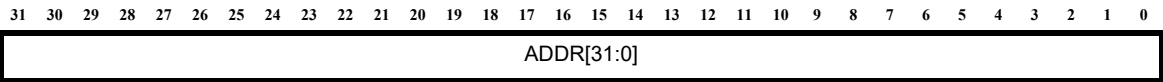
Field	Bits	Reset	Description
START	[31:0]	0	Filter's start address value

36.6.2.2 Trace Address Stop Filter Register, ARCT_ADSTP_FLTx

Access: RW

Reset: 0xFFFF_FFFF

Figure 36-20 Trace Address Stop Filter Register



This register holds the stop address for an address filter (range and trigger). The Range filter stops capturing the trace soon after the PC address value is matched or greater than the PC filter stop address.

-  **Note**
- Ensure that the stop address of a filter for program trace does not point to a delay-slot instruction.
 - If PC_SIZE is less than 32 bits, when programming this register, set the MSB [32:PC] bits to 0.

Table 36-24 Stop Address Register Field Descriptions

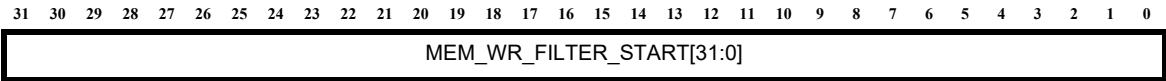
Field	Bits	Reset	Description
ADDR	[31:0]	0xFFFF_FFFF	Filter's stop address value

36.6.3 Memory Write Filter Start Register, MEM_WR_FILTER_START

Access: RW

Reset: 0x00

Figure 36-21 MEM_WR_FILTER_START Register



The Memory write filter start register holds the value of the start address of the memory for the range filter. The Range filter starts capturing the trace soon after the memory address value is matched or greater than the memory write filter start address. The start address must always be less than or equal to the filter stop address.

 **Note** If ADDR_SIZE is less than 32 bits, when programming this register, set the MSB [32:ADDR_SIZE] bits to 0.

Table 36-25 MEM_WR_FILTER_START Field Descriptions

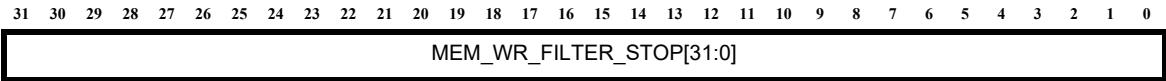
Field	Bits	Reset	Description
MEM_WR_FILTER_START	[31:0]	0	Holds the start address for memory write data filter.

36.6.4 Memory Write Filter Stop Register, MEM_WR_FILTER_STOP

Access: RW

Reset: 0x00

Figure 36-22 MEM_WR_FILTER_STOP Register



The Memory write filter stop register holds the value of the stop address of the memory for the range filter. The Range filter stops capturing the trace soon after the memory address value is matched or greater than the memory write filter stop address.

 **Note** If ADDR_SIZE is less than 32 bits, when programming this register, set the MSB [32:ADDR_SIZE] bits to 0.

Table 36-26 MEM_WR_FILTER_STOP Field Descriptions

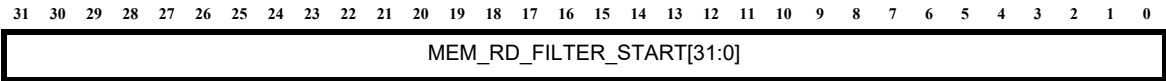
Field	Bits	Reset	Description
MEM_WR_FILTER_STOP	[31:0]	0	Holds the stop address for memory write data filter.

36.6.5 Memory Read Filter Start Register, MEM_RD_FILTER_START

Access: RW

Reset: 0x00

Figure 36-23 MEM_RD_FILTER_START Register



The Memory read filter start register holds the value of the start address of the memory for the range filter. The range filter starts capturing the trace soon after the memory address value is matched or the greater than the memory write filter start address. The start address must always be less than or equal to the filter stop address.

 **Note** If ADDR_SIZE is less than 32 bits, when programming this register, set the MSB [32:ADDR_SIZE] bits to 0.

Table 36-27 MEM_RD_FILTER_START Field Descriptions

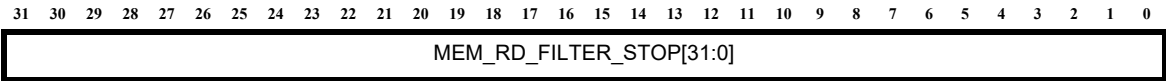
Field	Bits	Reset	Description
MEM_RD_FILTER_START	[31:0]	0	Holds the start address for memory read data filter.

36.6.6 Memory Read Filter Stop Register, MEM_RD_FILTER_STOP

Access: RW

Reset: 0x00

Figure 36-24 MEM_RD_FILTER_STOP Register



The Memory read filter stop register holds the value of the stop address of the memory for the range filter. The range filter stops capturing the trace soon after the memory address value is matched or greater than the memory write filter stop address.

 **Note** If ADDR_SIZE is less than 32 bits, when programming this register, set the MSB [32:ADDR_SIZE] bits to 0.

Table 36-28 MEM_RD_FILTER_STOP

Field	Bits	Reset	Description
MEM_RD_FILTER_STOP	[31:0]	0	Holds the stop address for memory read data filter.

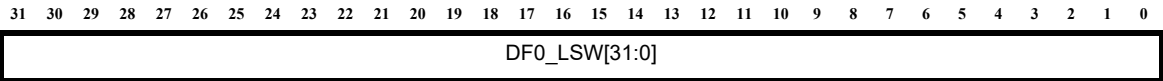
36.6.7 Data Filter0 LSW Register, PRDATAFILTER0_LSW

Access: RW

Offset from 0x10
0x040:

Reset: 0x0

Figure 36-25 Data Filter0 LSW Register



When data filter 0 is enabled, ARC Trace matches the source data with the data in this register. The “[Trace Filter Type Register, ARCT_FLT_TYPE](#)” on page 1153 indicates the source against which data filter 0 is programmed. When the data matches, ARC Trace starts capturing the trace messages. If this filter is configured for memory read or write trace, the data that is compared is dependent on the byte enables from the producer. ARC Trace uses the same byte enables as in the processor.

If your producer is configured for 64-bit memory operations, this register holds the lower 32-bits of the data to be compared.

This register is available only if your processor is configured as a medium producer.

Table 36-29 Data Filter0 LSW Description

Field	Bits	Reset	Description
DF0_LSW	[31:0]	0	Indicates the lower 32-bits of the data to be compared with a producer source.

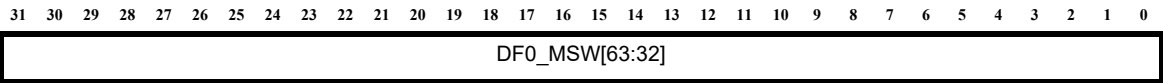
36.6.8 Data Filter0 MSW Register, PRDATAFILTER0_MSW

Access: RW

Offset from 0x40: 0x11

Reset: 0x0

Figure 36-26 Data Filter0 MSW Register



When data filter 0 is enabled, ARC Trace matches the source data with the data in this register. The “[Trace Filter Type Register, ARCT_FLT_TYPE](#)” on page 1153 indicates the source against which data filter 0 is programmed. When the data matches, ARC Trace starts capturing the trace messages. The data that is matched is dependent on the byte enables from the producer. ARC Trace uses the same byte enables as in the processor for matching data. For a 64-bit memory bus data width, this register holds the upper 32 bits of the data.



- This register exists only for medium producers with the following configuration: 64-bit memory data width or core register writes are 64-bit wide.

Table 36-30 Data Filter0 MSW Field Descriptions

Field	Bits	Reset	Description
DF0_MSW	[31:0]	0	Indicates the upper 32-bits of the data to be compared with a producer source.

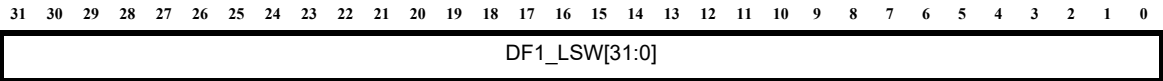
36.6.9 Data Filter1 LSW Register, PRDATAFILTER1_LSW

Access: RW

Offset from 0x40: 0x12

Reset: 0x0

Figure 36-27 Data Filter1 LSW Register



When data filter 1 is enabled, ARC Trace matches the source data with the data in this register. The “[Trace Filter Type Register, ARCT_FLT_TYPE](#)” on page 1153 indicates the source against which data filter 0 is programmed. When the data matches, ARC Trace starts capturing the trace messages. The data that is matched is dependent on the byte enables from the producer. ARC Trace uses the same byte enables as in the processor for matching data.

For a 64-bit memory bus data width, this register holds the lower 32 bits of the data.

This register is available only if your processor is configured as a medium producer.

Table 36-31 Data Filter1 LSW Field Descriptions

Field	Bits	Reset	Description
DF1_LSW	[31:0]	0	Indicates the lower 32-bits of the data to be compared with a producer source.

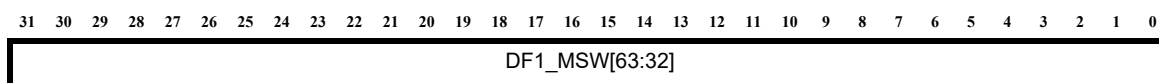
36.6.10 Data Filter1 MSW Filter, PRDATAFILTER1_MSW

Access: RW

Offset from 0x40: 0x13

Reset: 0x0

Figure 36-28 Data Filter1 MSW Filter



When data filter 1 is enabled, ARC Trace matches the source data with the data in this register. The “[Trace Filter Type Register, ARCT_FLT_TYPE](#)” on page 1153 indicates the source against which data filter 0 is programmed. When the data matches, ARC Trace starts capturing the trace messages. The data that is matched is dependent on the byte enables from the producer. ARC Trace uses the same byte enables as in the processor for matching data.

For a 64-bit memory bus data width, this register holds the upper 32 bits of the data.



- This register exists only for medium producers with the following configuration: 64-bit memory data width or core register writes are 64-bit wide.

Table 36-32 Data Filter1 MSW Description

Field	Bits	Reset	Description
DF1_MSW	[31:0]	0	Indicates the upper 32-bits of the data to be compared with a producer source.

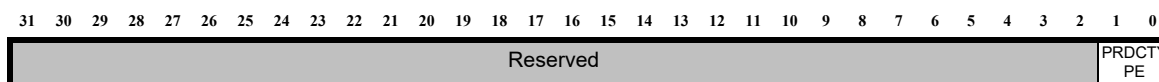
36.6.11 Trace Producer Type Register, ARCT_PRTYPE

Access: R

Offset from 0x40: 0x14

Reset: Inherited from the ARC EM build-time configuration (RTT Feature Level)

Figure 36-29 Producer Type Register



Holds information about the of the Producer ID and the type of producer messages that are generated.

Table 36-33 Producer Type Register Field Descriptions

Field	Bits	Reset	Description
PRDCTYPE	[1:0]	0	Producer Type Register. All non-enumerated values are reserved. 0x0: Small producer. Trace messages are suitable for program flow reconstruction. 0x1: Medium producer. Includes Trace messages of Small Producer type and Data memory transactions.
RESERVED	[31:2]	0	Reserved. Read-as-zero; ignored on write.

36.6.12 Trace Filter Type Register, ARCT_FLT_TYPE

Access: RW

Offset from 0x15
0x40:

Reset: 0x00; All filters are disabled

Figure 36-30 Filter Source Select Register

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								
DF1				DF0				FILTER 7				FILTER 6				FILTER 5				FILTER 4				FILTER 3				FILTER 2				FILTER 1				FILTER 0			

Filter type register selects the filters for the producer sources. Set of three bits of filter type register represents a range filter of a source. Multiple filters can be assigned to a single source by writing the address filter filed source selection value in the filter programming field.

Data filters enabled by writing the data filter filed source selection value in the filter programming field. For more information about filters, see [“Address Filter Registers”](#) on page 1140.

For memory operations, when the MSB (bit [27] for Data Filter0 and bit [31] for Data Filter1) is set to 1'b1, the data is compared on both read and write paths.

For core writes, when MSB (bit [27] for Data Filter0 and bit [31] for Data Filter1) is set to 1'b1, 64-bit data comparisons are enabled, otherwise 32-bit data comparisons are enabled.

Note: For EM, Memory data width is fixed to 32 bit. The core data filter type field's MSB (bit [27] for Data Filter0 and bit [31] for Data Filter1) is ignored.

Table 36-34 Filter Source Select Register Description

Field	Reset	Description
FILTER [n]	0	To assign a filter to a producer's source, the respective bits should be programmed as shown below. All non-enumerated values are reserved. 0x0 Filter is disabled 0x1 Program counter filter 0x2 Memory write filter (medium producer) 0x3 Memory read filter (medium producer) 0x7 : Global filter
DT[n]	0	Apply the data filter on a source 0x0 : Filter is disabled 0x2: Memory write 0x3: Memory read 0xA: Memory read and write 0xB: Memory read and write

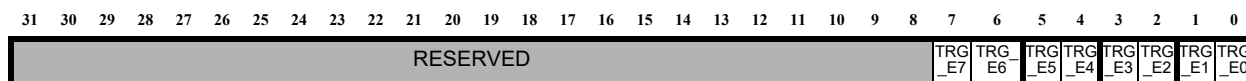
36.6.13 Trace Filter Control Register, ARCT_FLT_CTRL

Access: RW

Offset from 0x40: 0x16

Reset: 0x00

Figure 36-31 Trace Filter Control Register



Selects the filter as a range filter or a trigger filter.

Table 36-35 Trace Filter Control Register

Field	Bits	Reset	Description
TRG_E[0]	[0]	0	Producer filter 0 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[1]	[1]	0	Producer filter 1 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[2]	[2]	0	Producer filter 2 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[3]	[3]	0	Producer filter 3 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[4]	[4]	0	Producer filter 4 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[5]	[5]	0	Producer filter 5 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[6]	[6]	0	Producer filter 6 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.

Table 36-35 Trace Filter Control Register

Field	Bits	Reset	Description
TRG_E[7]	[7]	0	Producer filter 7 trigger enable ■ 1: Filter used as trigger filter. ■ 0: Filter used as range filter.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

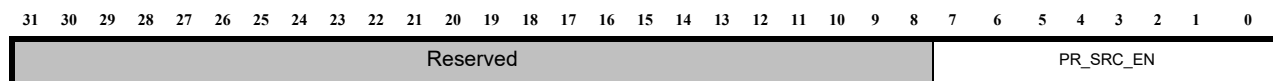
36.6.14 Trace Producer Source Enable Register, ARCT_PRSRC_E

Access: RW

Address 0x17
from 0x040:

Reset: 0x01

Figure 36-32 Producer Source Enable Register



Producer Source Enable register holds the active sources of the producer.

Table 36-36 Producer Source Enable Description

Field	Bits	Reset	Description
PR_SRC_EN	[7:0]	1	Enable a source in a producer - bit [0]: set this bit to enable program counter as a source. - bit [1]: set this bit to enable memory write data as a source. - bit [2]: set this bit to enable memory read data as a source. - bit [6]: set this bit to enable debug status messages. When this bit is enabled, ARC Trace generates the Debug Status Trace. All other values are reserved.
RESERVED	[31:8]	0	Reserved. Read-as-zero; ignored on write.

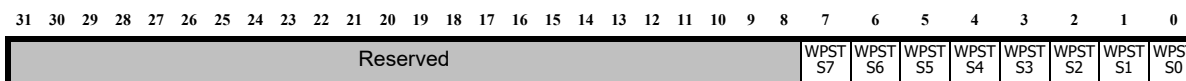
36.6.15 Trace Producer Watchpoint Status Register, ARCT_PRWP_STATUS

Access: R

Offset from 0x19
0x40:

Reset: 0x00

Figure 36-33 Producer Watchpoint Status Register



Each producer supports up to 8 watch points. The Producer actionpoints are mapped to the producer Watchpoint status register. The watch point messages are generated based on the bit set from the Watchpoint source of that producer.

Table 36-37 Producer Source Watchpoint Status Register Description

Field	Bits	Reset	Description
WPSTS0	[0]	0	Watchpoint Status 0 0x0: Producer actionpoint 0 de-asserted in this cycle 0x1: Producer actionpoint 0 asserted in this cycle
WPSTS1	[1]	0	Watchpoint Status 1 0x0: Producer actionpoint 1 de-asserted in this cycle 0x1: Producer actionpoint 1 asserted in this cycle
WPSTS2	[2]	0	Watchpoint Status 2 0x0: Producer actionpoint 2 de-asserted in this cycle 0x1: Producer actionpoint 2 asserted in this cycle
WPSTS3	[3]	0	Watchpoint Status 3 0x0: Producer actionpoint 3 de-asserted in this cycle 0x1: Producer actionpoint 4 asserted in this cycle
WPSTS4	[4]	0	Watchpoint Status 4 0x0: Producer actionpoint 4 de-asserted in this cycle 0x1: Producer actionpoint 4 asserted in this cycle
WPSTS5	[5]	0	Watchpoint Status 5 0x0: Producer actionpoint 5 de-asserted in this cycle 0x1: Producer actionpoint 5 asserted in this cycle

Table 36-37 Producer Source Watchpoint Status Register Description (Continued)

Field	Bits	Reset	Description
WPSTS6	[6]	0	Watchpoint Status 6 0x0: Producer actionpoint 6 de-asserted in this cycle 0x1: Producer actionpoint 6 asserted in this cycle
WPSTS7	[7]	0	Watchpoint Status 7 0x0: Producer actionpoint 7 de-asserted in this cycle 0x1: Producer actionpoint 6 asserted in this cycle
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

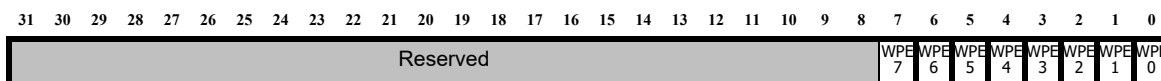
36.6.16 Trace Producer Watchpoint Enable Register, ARCT_PRWP_E

Access: RW

Offset from 0x1A
0x40:

Reset: 0x000000FF

Figure 36-34 Producer Watchpoint Enable Register



This register holds the enable for watchpoint messages when corresponding producer actionpoint is set in Producer Watchpoint status register.

Table 36-38 Producer Watchpoint Enable Register Description

Field	Bits	Reset	Description
WPE0	[0]	1	Enable Watchpoint 0 trace 0x0: Watchpoint trace 0 is disabled this cycle 0x1: Watchpoint trace 0 is enabled this cycle
WPE1	[1]	1	Enable Watchpoint 1 trace 0x0: Watchpoint trace 1 is disabled this cycle 0x1: Watchpoint trace 1 is enabled this cycle
WPE2	[2]	1	Enable Watchpoint 2 trace 0x0: Watchpoint trace 2 is disabled this cycle 0x1: Watchpoint trace 2 is enabled this cycle
WPE3	[3]	1	Enable Watchpoint 3 trace 0x0: Watchpoint trace 3 is disabled this cycle 0x1: Watchpoint trace 3 is enabled this cycle
WPE4	[4]	1	Enable Watchpoint 4 trace 0x0: Watchpoint trace 4 is disabled this cycle 0x1: Watchpoint trace 4 is enabled this cycle
WPE5	[5]	1	Enable Watchpoint 5 trace 0x0: Watchpoint trace 5 is disabled this cycle 0x1: Watchpoint trace 5 is enabled this cycle
WPE6	[6]	1	Enable Watchpoint 6 trace 0x0: Watchpoint trace 6 is disabled this cycle 0x1: Watchpoint trace 6 is enabled this cycle

Table 36-38 Producer Watchpoint Enable Register Description (Continued)

Field	Bits	Reset	Description
WPE7	[7]	1	Enable Watchpoint 7 trace ■ 0x0: Watchpoint trace 7 is disabled this cycle ■ 0x1: Watchpoint trace 7 is enabled this cycle
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

36.6.17 **Producer Cross Trigger Enable Register, CTIEN_REG_ADDR**

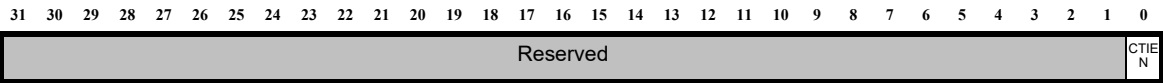
Access: RW

Offset from 0x1C

0x40:

Reset: 0

Figure 36-35 Producer Cross Trigger Enable Register, CTIEN_REG_ADDR



Set bit 0 to enable the external cross trigger interface trace start and stop inputs to control tracing. When enabled, these inputs override any global filters that have been set. This register exists in builds with the CoreSight offload option.

Table 36-39 Producer Cross Trigger Enable Register Description

Field	Bits	Reset	Description
CTIEN	[0]	0	Set bit 0 to 1 to enable the external trace start and stop control.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

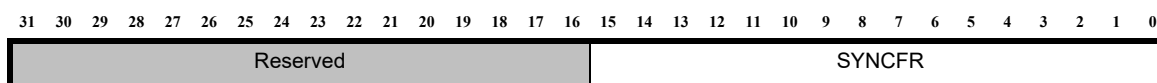
36.6.18 Producer ATB SYNC Frame Insertion Register, SYNCRFR_REG_ADDR

Access: RW

Offset from 0x1D
0x40:

Reset: 0x0000

Figure 36-36 Producer ATB SYNC Frame Insertion Register, SYNCRFR_REG_ADDR



This register sets the interval (in bytes transmitted) between insertion of SYNC frames (0xFF) into the CoreSight trace stream sent by the ARC Trace module into the CoreSight ATB infrastructure. These frames are used by the debugger to align the captured byte stream and allow reconstruction of the Nexus trace stream before processing. This register exists in builds with the CoreSight offload option.

Table 36-40 Producer SYNC Frame Insertion Register Description

Field	Bits	Reset	Description
SYNCRFR	[15:0]	0	Byte counter to set gap between SYNC frame insertion in ATB stream.
RESERVED	[31:16]	0	Reserved. Read-as-zero; ignored on write.

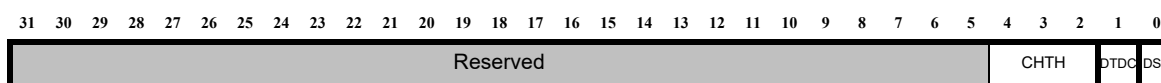
36.6.19 Producer Trace Attribute Register, ARCT_PRATTR

Access: RW

Offset from 0x40: 0x1E

Reset: 0x0

Figure 36-37 Producer Trace Attribute Register



Use this register to control the producer trace attributes such as: data suppression, including cache miss or hit information in a data trace message, and thresholds for generating a Resource Full message for conditional branches.

Table 36-41 Producer Trace Attribute Register Description

Field	Bits	Reset	Description
DSE	[0]	0	This bit is present only when ATB offload is configured. Control for data suppression feature, prioritizing program trace. Set to 1 to enable, 0 to disable. Use this bit to enable the data suppression feature. When this bit is set to 1 and both program and data tracing are enabled, data tracing is suppressed when the ATB FIFO reaches a defined threshold to try to prevent an overflow and subsequent loss of all trace information while preserving the program trace features.
DTDC	[1]	0	For ARC EM processors, always program this bit as 0. ■ Data Trace Data Config. This bit is available only for medium producers. ■
CHTH	[4:2]	0	Conditional History Threshold. Resource Full Messages issued for Conditional History overflow can be throttled, from the default setting, to allow finer grain instruction cycle calculations from the collected trace stream. For single-issue cores: ■ 0x2: RFM is generated for every 2nd branch ■ 0x3: RFM is generated for every 4th branch ■ 0x4: RFM is generated for every 8th branch ■ 0x5: RFM is generated for every 16th branch ■ 0x6: RFM is generated for every 16th branch ■ 0x7: RFM is generated for every 16th branch All other values: RFM is generated every 29th branch

Table 36-41 Producer Trace Attribute Register Description (Continued)

Field	Bits	Reset	Description
RESERVED	[31:5]	0	Reserved. Read-as-zero; ignored on write.

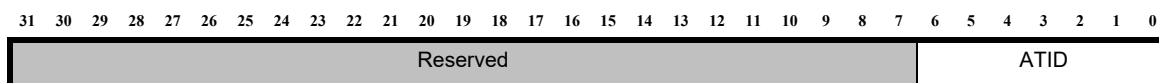
36.6.20 Producer CoreSight ID Register, ATID_REG_ADDR

Access: RW

Offset from 0x1F
0x40:

Reset: 0x0001

Figure 36-38 Producer CoreSight ID Register



Sets the ID on the CoreSight infrastructure for this producer. Each trace source in the system must be set to have a unique ID value.

Table 36-42 Producer CoreSight ID Register Description

Field	Bits	Reset	Description
ATID	[6:0]	1	[6:0] ID field output into ATB infrastructure for this producer
RESERVED	[31:7]	0	Reserved. Read-as-zero; ignored on write.

37

Power Management Features

37.1 Introduction to Power Management Features

The ARCV2 ISA provides a number of features to control power consumption, depending on the configuration of the core. These features may include power domain management (PDM), to reduce static (leakage) power through the power-down of unused components and features to enable an external controller to dynamically adjust the supply voltage and operating frequency of the core (DVFS).

37.2 PDM and DVFS Register Interface

Table 37-1 Power Domain Management Registers

Address	Name	Description
0x610	Core Power Status Register, PDM_PSTAT	Core power status; this register is present only when PDM is configured.
0x611	ARC Trace Power Status Register, RTT_PDM_PSTAT	ARC RTT power status; this register is present only when ARC RTT and PDM are configured.
0x612	Core Performance Register, DVFS_PL	Current performance level of the core; this register is present only when DVFS is configured.
0x613	Power Down Register, PDM_PMODE	Core power down programming register; this register is present only when PDM is configured.
0xF7	Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD	Build configuration register

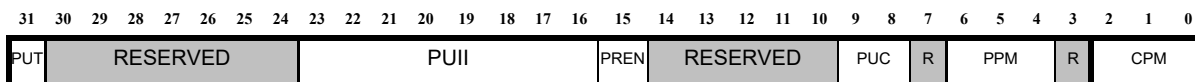
37.2.1 Core Power Status Register, PDM_PSTAT

Address: 0x610

Access: When SecureShield 2+2 mode is not configured: RW
When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x0000_0000

Figure 37-1 PDM_PSTAT Register



This register indicates the current power mode of the core, the previous power mode of the core, the power up cause, and the interrupt index that has caused the power up of the core. The debugger can set the PUT bit of this register to power up the core. In earlier releases, this register was named `DEBUGP`.

Table 37-2 PDM_PSTAT Field Description

Field	Bit	Reset	Description
CPM	[2:0]	0	Current power mode of the core. Read-only. This field can only be controller by the hardware. Write to this field are ignored. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode For more information about the power-down modes, see Table 37-5 on page 1172.
PPM	[6:4]	0	Previous power mode of the core. Read-only. This field can only be controller by the hardware. Write to this field are ignored. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode For more information about the power-down modes, see Table 37-5 on page 1172.

Table 37-2 PDM_PSTAT Field Description

Field	Bit	Reset	Description
PUC	[9:8]	0	Power up cause. Read-only. This field is relevant only if the core was previously in a power-down mode. 0x0: Power up by interrupt; for more information about the interrupts that can power up, see the <i>DesignWare ARCV2-based processor Databook</i>. 0x1: Power up by debugger 0x2: Power up by an external event This field can only be controlled by hardware. Writes to this field are ignored.
PREN	[15]	0	Previous memory power status in PM2 mode. Read only. 0x0: PD2-ICCM-R shut down 0x1: PD2-ICCM-R retention In PM1 mode, this field is ignored and return value is 0. This field can only be controlled by hardware. Writes to this field are ignored.
PUII	[23:16]	0	This field is relevant only if the core was previously in a power-down mode and was powered up by an interrupt. The index of the power up interrupt. Value range is from 16 to 255. Read-only.
PUT	[31]	0	Power up trigger bit. This bit is write-only and is always read as zero. - Write 1: - When core is in power-down state: Send power up request to the PMU and halt the core after PMU has powered up the core. - When core is in power on state: halt the core. - Write 0: No effect; ignored.

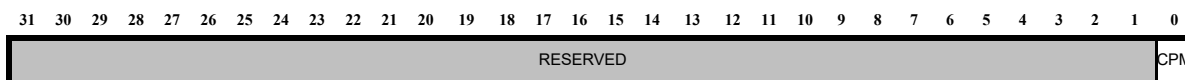
37.2.2 ARC Trace Power Status Register, RTT_PDM_PSTAT

Address: 0x611

Access: RW

Reset: 0x0000_0000

Figure 37-2 RTT_PDM_PSTAT Register



Use this register to power down or power up ARC Trace. Reading this register returns the current ARC RTT power status.

Table 37-3 RTT_PDM_PSTAT Field Description

Field	Bit	Reset	Description
CPM	[0]	0	Write to this bit to power down or power up ARC Trace 0x0: Send ARC Trace power up request to the external PMU. 0x1: Send ARC Trace power down ARC Trace to the external PMU in PM1 mode. Read this bit to determine the power status of ARC Trace: 0x0: The external PMU has powered up ARC Trace. 0x1: The external PMU has powered down ARC Trace.

37.2.3 Core Performance Register, DVFS_PL

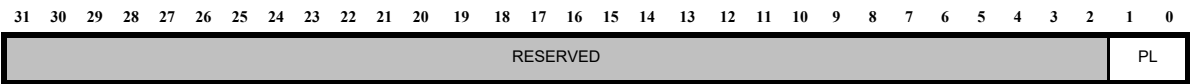
Address: 0x612

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Default: 0x0000_0000

Figure 37-3 DVFS_PL Register



Write to this register to set the performance level of a core. When you write to this register, a request is sent to the external DVFS controller to change the performance level accordingly.

Table 37-4 DVFS_PL Field Description

Field	Bit	Description
PL	[1:0]	Current performance level of a core. 0x0: PL0 (performance level 0). This is the default value. 0x1: PL1 (performance level 1) 0x2: PL2 (performance level 2) 0x3: PL3 (performance level 3) For more information about the performance levels, see the <i>DesignWare ARC EM Databook</i>.

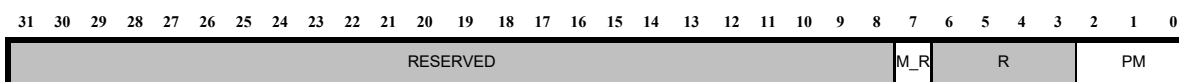
37.2.4 Power Down Register, PDM_PMODE

Address: 0x613

Access: When SecureShield 2+2 mode is not configured: W
When SecureShield 2+2 mode is configured: W in the secure mode only

Default: 0x0000_0000

Figure 37-4 PDM_PMODE Register



Use this register to program the power-down mode of a core. Write to this register before executing the SLEEP instruction. This register is cleared to the default value, 0, after the core is powered down.

For information about the power domains, see the *DesignWare ARCV2-based Processor Databook*.

Table 37-5 PDM_PMODE Field Description

Field	Bit	Reset	Description
PM	[2:0]	0	Power-down mode ■ 0x0: power on; default ■ 0x1: power-down mode 1 (PM1) ■ 0x2: power-down mode 2 (PM2) ■ 0x3: power-down mode 3 (PM3)
M_RETEN	[7]	0	Memory power down status ■ 0x0: First partition of ICCM0 is shut down ■ 0x1: First partition of ICCM0 is in retention Use this field program the ICCM0-R memory power status in the PM2 mode. If you are programming the core to be in the PM2 mode, ensure that you also program the this field (M_RETEN) field before executing the SLEEP instruction. This field is ignored in the PM1 mode. The best practice is to program this field to 0 in the PM1 mode.

37.3 Build Configuration Registers

The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

A set of baseline Build Configuration Registers are present in all ARCV2 processors. In addition, each optional ISA extension typically includes a Build Configuration Register to signify its presence and its specific configuration in each given build.

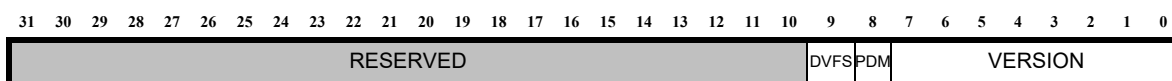
Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCV2-based system.

37.3.1 Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD

Address: 0xF7

Access: R

Figure 37-5 PDM_DVFS_BUILD Register



This register indicates the presence of the power-domain management and DVFS capabilities in an ARCV2-based processor.

Table 37-6 PDM_DVFS_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	0x2: initial version 0x5: indicates support for the ICCM0 partition
PDM	[8]	Indicates whether the processor power consumption can be controlled by dividing the processor into different power domains. 0x1: The processor can be divided into different power domains. 0x0: The processor is not divided into different power domains.
DVFS	[9]	Indicates whether the DVFS (dynamic voltage and frequency scaling) functionality is supported. 0x1: DVFS is supported. 0x0: DVFS is not supported.

	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
r0	Basecase Core Registers																															
r1																																
-																																
r24																																
r25																																
r26	Global Pointer (GP)																															
r27	Frame Pointer (FP)																															
r28	Stack Pointer (SP)																															
r29	Interrupt Link Register (ILINK)																															
r30	General Purpose Register																															
r31	Branch Link Register (BLINK)																															
r32																																
-	Extension Core Registers																															
r57																																
r58	Accumulator Low ACCL (little endian), ACCH (big-endian)																															
r59	Accumulator High- ACCH (little endian), ACCL (big endian)																															
r60	LP_COUNT[LPC_SIZE-1:0]																															
r61	RESERVED																															

Figure 38-1 Core Register Map Summary (Continued)

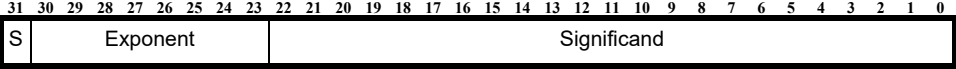
r62	Long Immediate Data Indicator														
r63	Program Counter [PC_SIZE-1:2], Read-Only (PCL)													0	0

38.2 Extension Core Registers

The register set is extensible in register positions r32 to r57.

When a floating-point unit is included and the FSMADD and FSMSUB instructions are enabled, the two extension core registers (r58 and r59) are included and comprise the third operand as follows:

Figure 38-2 ACCL for fusedMultiplyAdd and fusedMultiplySubtract Operations



 **Note**

The registers r58 and r59 are reserved and cannot be defined by APEX extensions. However, these registers are accessible (explicitly) by APEX instructions if the processor is configured to implement these registers.

If R58/R59 are used as a destination in a multiply and accumulate instruction, the update of the accumulator takes precedence over the write-back. In this case the write-back value is discarded.

38.3 FPU Auxiliary Registers

Table 38-1 FPU Registers

Address	Auxiliary Register Name	Description
0xC8	Floating-Point Unit Build Register, FPU_BUILD	Build configuration register
0x300	Floating-Point Unit Control Register, FPU_CTRL	Contains controls that let you mask floating-point exceptions and control the rounding behavior of floating-point operations
0x301	Floating-Point Unit Status Register, FPU_STATUS	Contains the sticky flags to indicate the status of floating-point operations and controls for resetting the sticky flags
0x302	Double-precision Floating Point D1 Lower Register, AUX_DPFP1L	Double-precision floating-point register D1. This register is present only if FPU_DP_ASSIST configuration option is enabled.
0x303	Double-precision Floating Point D1 Higher Register, AUX_DPFP1H	Double-precision floating-point register D1. This register is present only if FPU_DP_ASSIST configuration option is enabled.
0x304	Double-precision Floating Point D2 Lower Register, AUX_DPFP2L	Double-precision floating-point register D2. This register is present only if FPU_DP_ASSIST configuration option is enabled.
0x305	Double-precision Floating Point D2 Higher Register, AUX_DPFP2H	Double-precision floating-point register D2. This register is present only if FPU_DP_ASSIST configuration option is enabled.

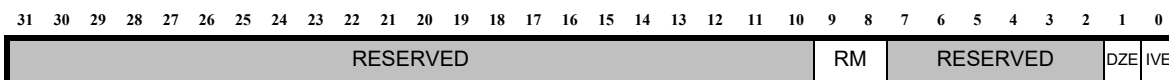
38.3.1 Floating-Point Unit Control Register, FPU_CTRL

Address: 0x300

Access: rw

Default 0x00000100

Figure 38-3 FPU_CTRL Register



This register controls the behavior of the single-precision floating-point unit in response to events such as a division-by-zero, invalid operation, or rounding of a result.

Table 38-2 FPU_CTRL Field Description

Field	Bits	Reset	Description
IVE	0	0	Enable Invalid Operation Exception 0x0: Exception is not raised when an invalid operation occurs and the IV flag in the FPU_STATUS register is set. 0x1: Exception is raised when an invalid operation occurs and the IV flag in the FPU_STATUS register is not set. The double-precision floating-point accelerator instructions (DADDx, DSUBx, DMULx, DEXCLx) do not raise this exception.
DZE	1	0	Enable Divide by Zero Exception 0x0: Exception is not raised when a divide by zero operation occurs and the DZ flag in the FPU_STATUS register is set. 0x1: Exception is raised when a divide by zero operation occurs and the DZ flag in the FPU_STATUS register is not set. This exception is raised only by the hardware divider implementation. A software divider implementation does not raise this exception.
RESERVED	[7:2]	0	Reserved. Read-as-zero; ignored on write.

Table 38-2 FPU_CTRL Field Description

Field	Bits	Reset	Description
RM	[9:8]	1	Indicates Rounding Mode ■ 0x0: Round towards zero; the precise result is rounded to the nearest representable number closest to zero. ■ 0x1: Round to nearest even (default); the precise result is rounded to the nearest representable number. If the precise result is exactly between two representable numbers, the result is rounded to the even representable number. ■ 0x2: Round to positive infinity; the result is rounded to the next most positive representable number. ■ 0x3: Round to negative infinity; the result is rounded to the next most negative representable number.
RESERVED	[31:10]	0	Reserved. Read-as-zero; ignored on write.

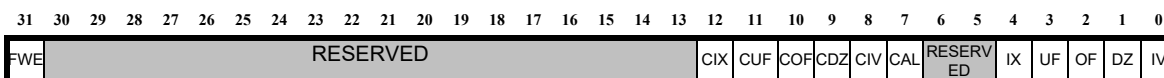
38.3.2 Floating-Point Unit Status Register, FPU_STATUS

Address: 0x301

Access: rw

Reset 0x00000000

Figure 38-4 FPU_STATUS Register



This register allows you to read the status of the floating-point unit flags, and clear any set flags. These flags are updated based on the result of a single-precision floating-point operation. On processor reset, all the flags are cleared.

Table 38-3 FPU_STATUS Field Description

Field	Bits	Reset	Description
IV	0	0	Invalid operation sticky flag. This flag is updated implicitly based on the result of a floating-point operation. This flag is a sticky flag. You need to explicitly write to the CIV bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the CIV bit to clear this flag. ■ If FWE==1, this bit has both read and write access.
DZ	1	0	Divide-by-zero sticky flag. This flag is updated implicitly based on the result of a floating-point operation. This flag is a sticky flag. You need to explicitly write to the CDZ bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the CDZ bit to clear this flag. ■ If FWE==1, this bit has both read and write access.
OF	2	0	Overflow sticky flag. This flag is updated implicitly based on the result of a floating-point operation. This flag is a sticky flag. You need to explicitly write to the COF bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the COF bit to clear this flag. ■ If FWE==1, this bit has both read and write access.

Table 38-3 FPU_STATUS Field Description

Field	Bits	Reset	Description
UF	3	0	Underflow sticky flag. The underflow is checked and the underflow flag is updated before the result of the floating-point instruction is rounded (before rounding as declared in the std IEEE Std 754-2008). You need to explicitly write to the CUF bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the CUF bit to clear this flag. ■ If FWE==1, this bit has both read and write access.
IX	4	0	Inexact sticky flag. This flag is updated implicitly based on the result of a floating-point operation. This flag is a sticky flag. You need to explicitly write to the CIX bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the CIX bit to clear this flag. ■ If FWE==1, this bit has both read and write access.
RESERVED	[6:5]	0	Reserved. Read-as-zero; ignored on write.
CAL	7	0	Clear all flags. ■ If FWE==0, you can write to this bit. Write 1 to clear bits [4:0]. ■ If FWE==1, this bit read as zero and ignored on write.
CIV	8	0	Clear invalid operation flag. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the IV bit. ■ If FWE==1, this bit read as zero and ignored on write.
CDZ	9	0	Clear divide by zero. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the DZ bit. ■ If FWE==1, this bit read as zero and ignored on write.
COF	10	0	Clear overflow flag. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the OF bit. ■ If FWE==1, this bit read as zero and ignored on write.
CUF	11	0	Clear underflow flag. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the UF bit. ■ If FWE==1, this bit read as zero and ignored on write.
CIX	12	0	Clear inexact flag. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the IX bit. ■ If FWE==1, this bit read as zero and ignored on write.

Table 38-3 FPU_STATUS Field Description

Field	Bits	Reset	Description
RESERVED	[30:11]	0	Reserved. Read-as-zero; ignored on write.
FWE	31	0	Flag Write Enable 0x0 : Enables write only access to bits [12:7]; bits[4:0] are read-only and writes are ignored; other bits are ignored on writes and read as zero. 0x1 : Enable write access to bits [4:0], these bits can be read-only if FWE==0. Other bits are read as zero and ignored on write.

IV – Invalid Operation

The invalid operation flag is set in the following cases:

- If any of the input operands to a floating-point instruction is a signaling NaN when `FPU_CTRL.IVE==0` (invalid operation exceptions is disabled).
- For FCVT32: Set when a floating-point number is converted to an integer, where the floating-point number is NaN, or infinity, or the conversion result is too big to be represented as an integer.
- For FSADD: Set when the operands are in the following combinations: (b= +infinity and c = -infinity) or (b= -infinity and c = +infinity); or when an input operand is a signaling NaN.
- For FSCMP: Set when either of the inputs is a signaling NaN and the comparison result is unordered.
- For FSCMPF: Set when either of the inputs is a quiet or signaling NaN and the result is unordered.
- For FSDIV: Set for division operations of 0/0 and infinity/(infinity); or when an input operand is a signaling NaN.
- For FSMUL, FSMADD, FSMSUB: Set when multiplication of 0 x (infinity) or (infinity) x 0 occurs; or when an input operand is a signaling NaN.
- For FSSQRT: Set when the c operand is less than -zero; or when an input operand is a signaling NaN.
- For FSSUB: Set when the operands are both +infinity or both are -infinity; or when an input operand is a signaling NaN.

DZ – Divide-by-Zero

The divide-by-zero flag is set when the divisor is zero (0), and the dividend is a finite non-zero number.

OF – Overflow

Overflow occurs when the result is too large to be represented by the instruction data format. The result is rounded and then saturated.

UF – Underflow

Underflow occurs when the result is too small to be represented by the instruction data format. The underflow flag is set before rounding. The result is rounded and then saturated.

IX — Inexact

Inexact operation occurs when the result of a floating-point operation is rounded and not exact.

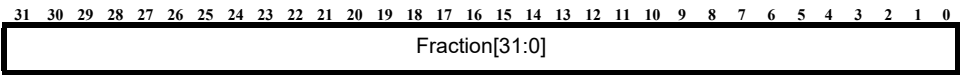
38.3.3 Double-precision Floating Point D1 Lower Register, AUX_DPFP1L

Address: 0x302

Access: RW

Reset 0x00000000

Figure 38-5 AUX_DPFP1L, Double-precision Floating Point D1 Register, Lower Half



The AUX_DPFP1L and AUX_DPFP1H registers represent the first 64-bit data register, D1, used by the double-precision floating point instructions. The AUX_DPFP1L register represents the lower half, bits [31:0], of D1 “Double-Precision Floating-Point Data Formats”.

Table 38-4 AUX_DPFP1L Field Description

Field	Bit	Description
Fraction	[31:0]	Fractional part of the floating point number

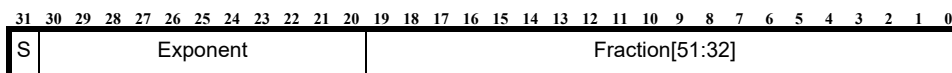
38.3.4 Double-precision Floating Point D1 Higher Register, AUX_DFP1H

Address: 0x303

Access: RW

Reset 0x00000000

Figure 38-6 AUX_DFP1H, Double-precision Floating Point D1 Register, Higher Half



The `AUX_DFP1L` and `AUX_DFP1H` registers represent the first 64-bit data register, D1, used by the double-precision floating point instructions. The `AUX_DFP1H` register represents the higher half, bits [63:32], of D1 “Double-Precision Floating-Point Data Formats”.

Table 38-5 AUX_DFP1H Field Description

Field	Bit	Description
Fraction	[19:0]	Fractional part of the floating point number [51:32]
Exponent	[30:20]	Exponent part of the floating point number
S	[31]	Sign part of the floating point number 0x0: Positive number 0x1: Negative number

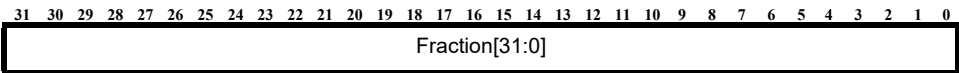
38.3.5 Double-precision Floating Point D2 Lower Register, AUX_DPFP2L

Address: 0x304

Access: RW

Reset 0x00000000

Figure 38-7 AUX_DPFP2L, Double-precision Floating Point D2 Register, Lower Half



The AUX_DPFP2L and AUX_DPFP2H registers represent the second 64-bit data register, D2, used by the double-precision floating point instructions. The AUX_DPFP2L register represents the lower half, bits [31:0] of D2 “[Double-Precision Floating-Point Data Formats](#)”.

Table 38-6 AUX_DPFP2L Field Description

Field	bit	Description
Fraction	[31:0]	Fractional part of the floating point number

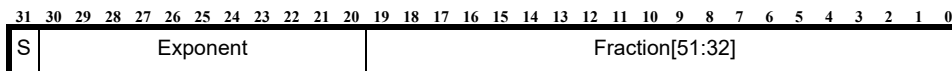
38.3.6 Double-precision Floating Point D2 Higher Register, AUX_DPFP2H

Address: 0x305

Access: RW

Reset 0x00000000

Figure 38-8 AUX_DPFP2H, Double-precision Floating Point D2 Register, Higher Half



The AUX_DPFP2L and AUX_DPFP2H registers represent the second 64-bit data register, D2, used by the double-precision floating point instructions. The AUX_DPFP2H register represents the higher half, bits [63:32], of D2 “[Double-Precision Floating-Point Data Formats](#)”.

Table 38-7 AUX_DPFP2H Field Description

Field	Bit	Description
Fraction	[19:0]	Fractional part of the floating point number [51:32]
Exponent	[30:20]	Exponent part of the floating point number
S	[31]	Sign part of the floating point number 0x0: Positive number 0x1: Negative number

38.4 Build Configuration Registers

The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

A set of baseline Build Configuration Registers are present in all ARCV2 processors. In addition, each optional ISA extension typically includes a Build Configuration Register to signify its presence and its specific configuration in each given build.

Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCV2-based system.

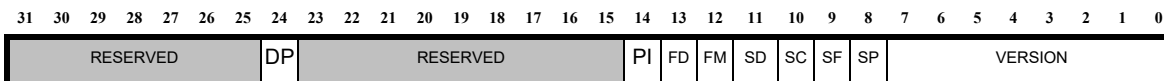
Auxiliary registers, in the range 0x60 to 0x7F and 0xC0 to 0xFF, are assumed to be BCRs. In kernel mode, any read from a non-existent build configuration register in this range returns 0, and no exception is generated. This design enables the kernel-mode code to detect the presence or absence of a BCR because all BCRs that are present in a system contain non-zero values.

38.4.1 Floating-Point Unit Build Register, FPU_BUILD

Address: 0xC8

Access: R

Figure 38-9 FPU_BUILD Register



This register is present only if a floating-point unit is included in the processor configuration. [Table 38-8](#) lists the field descriptions:

Table 38-8 FPU_BUILD Field Descriptions

Field	Bits	Description
VERSION	[7:0]	Floating-Point Unit Version 0x2: Initial version. With this version, support for single-precision is always true. 0x3: Current version. With this version, the pipelined floating-point unit implementation is supported.
SP	8	Indicates support for single-precision floating-point instructions.
SF	9	Indicates support for single-precision fusedMultiplyAdd and fusedMultiplySubtract instructions.
SC	10	Indicates support for single-precision floating-point format conversion instructions.
SD	11	Indicates support for single-precision square-root and divide instructions.
FM	12	Indicates support for fast multiplication instructions. 0: 2 cycle implementation 1: 1 cycle implementation
FD	13	Indicates support for fast divide and square root instructions. 0x0: 17 cycle implementation of the divide and square root instructions 0x1: 1 cycle implementation of the divide and square root instructions
PI	14	Indicates floating-point unit pipelined implementation 0x0: non-pipelined implementation 0x1: pipelined implementation; this implementation must be used with multi-cycle floating-point instructions only.

Table 38-8 FPU_BUILD Field Descriptions

Field	Bits	Description
DP	24	Indicates support for double-precision floating-point instructions

39

XY

The XY Memory is a feature commonly found in DSP processors to increase the DSP performance. The XY component allows a processor to implicitly load source operands and store results into a closely coupled memory using a single instruction. This results in dense code and high performance. The XY memory is accessed through a set of address pointers and modifiers in the address generation unit (AGU). Modifiers define the data type and increment values for the XY address pointers in the AGU. For the ARCV2 cores, X memory is often denoted as XCCM, and Y memory is often denoted as YCCM.

The XY memory architecture allows a single ARCV2 instruction to perform the following operations in a single clock cycle:

- Load two source operands (one from X memory and one from Y memory) from XY memory by dereferencing address pointers.
- Post-increment the address pointers.
- Compute the result of the instruction (multiply the operands, MAC the operands etc).
- Write back the result to XY memory by de-referencing an address pointer.
- Post-increment the destination address pointer.

39.1 XY Data Organization

In an ARC processor with XY memories, the data resides in the one of the following locations:

- External memory accessed through a cache or a load and store queue.
- One physical memory that includes DCCM, X, and Y.
- Two physical memories: one for DCCM+X and one for Y.
- Three physical memories: one each for DCCM, X, and Y.

To efficiently support an unaligned access, the logical memories can be split into two physical memory instances that can be interleaved in terms of accesses, one storing even data, the other storing odd data. Splitting the memories provides a pseudo dual-port memory implementation such that the processor and a DMA can access the same bank in a single cycle. Interleaving of the DCCM is controlled through the `dccm_interleave` option and the interleaving of the X and Y memories is controlled through the `xy_interleave` option.

The XCCM and YCCM can be associated with one of the memory regions in the range: 8 to 15. Memory regions of different memories should not overlap. If memories overlap, that is, they occupy the same region, DCCM accesses mask accesses to YCCM which will mask accesses to XCCM. The XCCM and YCCM can be associated with any of the 16 equally sized memory regions within the memory map. If a CCM is smaller than the associated memory region, the CCM is mirrored across the full memory region. If a CCM is larger than a region, it covers multiple regions. The base address of the CCMs should be aligned to the size of the CCM.

For example, in a 24-bit processor, the addressable memory space is 256 MB. If the processor is configured for 16 regions, each region is 16 MB. If the X memory is 64 KB at base address 0x900000, the last address of the X memory is 0x90FFFF. The first X memory mirror resides at the aperture: 0x910000 to 0x91FFFF; the last mirror resides at the aperture 0x9F0000 to 0x9FFFFF near the end of the memory region.

Code checking is not applied to the XY address generation unit (AGU), as the ICCM cannot be accessed from the AGU; stack checking is not applied to AGU because addressing is not stack-pointer relative.

Unaligned accesses crossing the boundary between ICCM and DCCM/XCCM/YCCM generate an unaligned exception, even if the processor supports unaligned load and store. Similarly, accesses crossing the boundary between a CCM and an memory generate an unaligned memory exception.

Any address outside the DCCM, XCCM, and YCCM memory regions cannot be used in an address generator. If used, an exception is raised.

The XY option requires that the DCCM is present in the processor build. If the DCCM is configured to include a DMI interface, the XY memory banks also include a DMI interface.

39.1.1 Address Generation

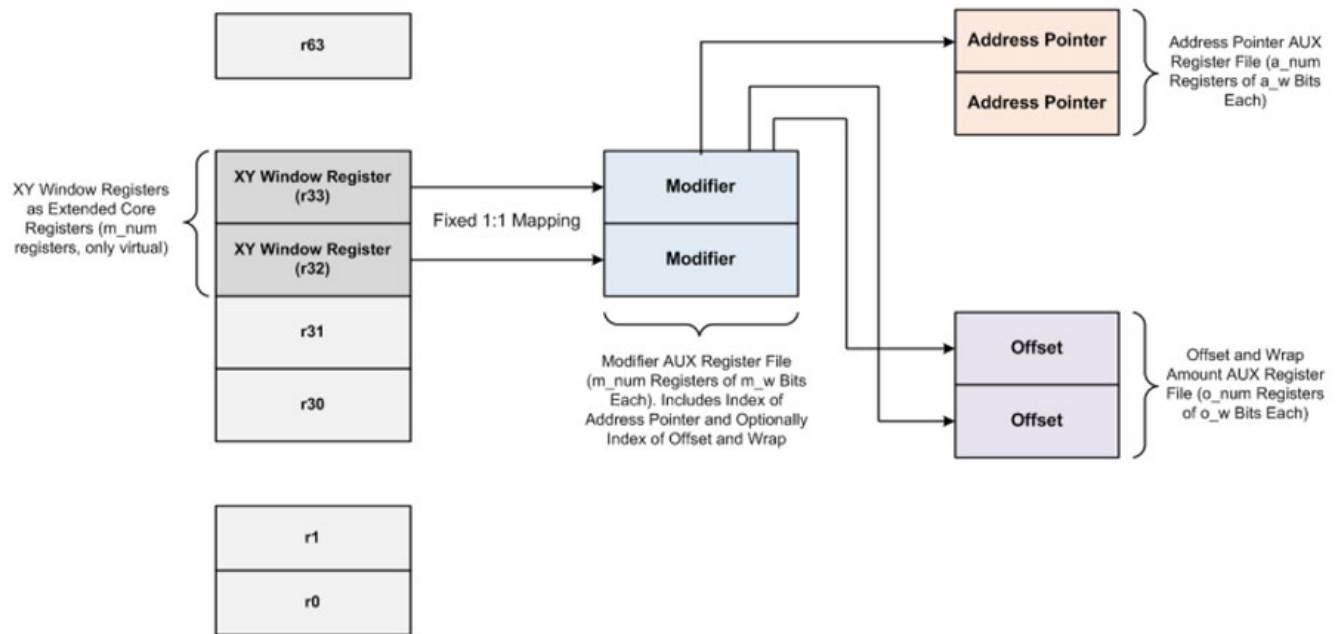
The ARC EM processor provides an implicit memory access to the XY memory through the address generation unit (AGU) as well as explicit access through load and store operations. If you have included an AGU in your processor, and an instruction tries to access the XY memory or DCCM, the processor uses the AGU to calculate the memory address.

Instructions access the XY memory or DCCM by referring to the AGU window registers that are mapped to extended core registers. Each window register has a corresponding AGU modifier register in the auxiliary register space. The following register sets are used to calculate the memory addresses in the XY memory.

- **Modifier register:** A modifier register has several fields describing the AGU memory access attributes. A modifier register includes an attribute freely associating the modifier with an address pointer register.
- **Address-pointer registers:** The address-pointer registers point to an arbitrary byte location in the XCCM, YCCM, or DCCM memory.
- **Offset registers:** A modifier register includes an offset value defining the post-increment value for the address pointer; the modifier offset value can be an immediate or a reference to a value in an arbitrary offset register; optionally a modifier may contain a wrap boundary attribute referring to a wrap value specified in an offset register.

Figure 39-1 illustrates the AGU register organization.

Figure 39-1 AGU Register Data Organization



The following steps are involved in the address generation process:

1. When an instruction is decoded, a modifier register is used to calculate the address.
2. The modifier register uses an address pointer register to access the XY memory or DCCM locations.
3. The next memory address is calculated by post-incrementing an address pointer in the following ways:
 - Increment the address pointer using the offset register value or the offset immediate value.
 - Bit-reverse increment of the address pointer using the offset register value or the offset immediate value. Bit-reversing involves the following steps:
 - i. Reverse the bits, that is, swap the MSB bits for the LSB bits, in the address pointer register and the offset register.
 - ii. Increment the address pointer using the offset value.

- iii. Reverse the bits again in the address pointer register.
- Modulo increment and wrapping: An increment region is created in the XY memory or DCCM with the following boundaries:
 - lower boundary==base address of the XY memory region
 - upper boundary==lower boundary plus the wrap value.

When the address pointer is incremented with the offset (register or immediate), it has to remain in the increment region. If the address pointer post increment crosses the upper boundary, the address pointer wraps to the lower boundary. If an address is negative, the address pointer wraps to the upper boundary.

39.1.2 Data Transformation

XY memory accesses support data transformations such as replication and sign extension. The AGU controls the kind of data transformations you can perform on the XY data.

The data transformation uses the following terminology:

- Data: information coming from the memory.
- Container: Expanded data. Data may be expanded into a wider container.
- Vector: One or multiple containers.

The following data transformations can be performed on the source operands:

- Resize the data from XY memory to align with MSB or LSB while zero-extending or sign-extending the data.
- Replicate a container across the width of a vector.
- Reverse the order of a container.

The destination operands support the following data transformations:

- Limit the width of the write-back operation.
- Swap data in containers (for example, endian conversion).
- Reverse vector container order (for example, endian conversion).

Related Topics:

- "XY Extension Core Registers"
- "XCCM_BASE"
- "YCCM_BASE"
- "AGU Address Pointer Registers, AGU_AUX_APx"
- "AGU Offset Registers, AGU_AUX_OSx"
- "AGU Modifier Registers, AGU_AUX_MODx"

39.2 XY Extension Core Registers

If the `HAS_AGU` option is true, the extended core registers `r32` and onwards are used as the AGU window registers. The number of AGU window registers is determined by the `AGU_MOD_NUM` AGU option. The window registers are available as extended core registers `rX`, where $32 \leq X < 32 + \text{AGU\_MOD\_NUM}$.

Reading or writing to these window registers indirectly accesses the XY memory while updating the address pointers.



Note

The configuration option, `AGU_SIZE`, controls the number of window registers present in a core. Any unassigned AGU window registers can be used by APEX.

Figure 39-2 Extended Core Registers when HAS_AGU==true

	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
r0																																
r1																																
-																																
r24																																
r25																																
r26	Global Pointer (GP)																															
r27	Frame Pointer (FP)																															
r28	Stack Pointer (SP)																															
r29	Interrupt Link Register (ILINK)																															
r30	General Purpose Register																															
r31	Branch Link Register (BLINK)																															
r32																																
-	AGU Window Registers																															
r57																																
r58	Accumulator Low ACCL (little endian), ACCH (big-endian)																															
r59	Accumulator High- ACCH (little endian), ACCL (big endian)																															
r60	LP_COUNT[LPC_SIZE-1:0]																															
r61	RESERVED																															
r62	Long Immediate Data Indicator																															
r63	Program Counter [PC_SIZE-1:2], Read-Only (PCL)																													0	0	

39.3 XY Register Set

Table 39-1 XY Registers

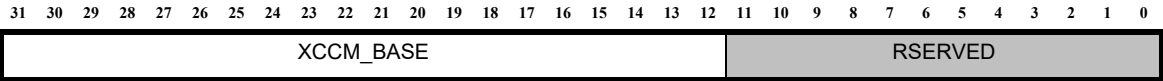
Address	Name	Description
0x5F8	XCCM Base Address, XCCM_BASE	XCCM base address
0x5F9	YCCM Base Address, YCCM_BASE	YCCM base address
0x79	XY Build Configuration Register, XY_BUILD	XY Build Configuration Register

39.3.1 XCCM Base Address, XCCM_BASE

Address: 0x05F8

Access: RW

Figure 39-3 XCCM_BASE Register



If the X memory is configured, this register indicates the memory region where the X memory is mapped in memory. If the X memory is not configured, this register does not exist. The X base address needs to be aligned to the size of a memory region and to the size of the memory itself. The X memory can be mapped to one of the memory regions from 8 to 15.

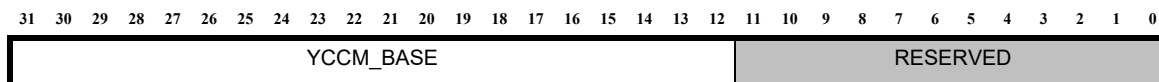
- If the address bus width is 16, the fields [15:12] indicate the XCCM memory region.
- If the address bus width is 20, the fields [19:16] indicate the XCCM memory region.
- If the address bus width is 24, the fields [23:20] indicate the XCCM memory region.
- If the address bus width is 32, the fields [31:28] indicate the XCCM memory region.

39.3.2 YCCM Base Address, YCCM_BASE

Address: 0x5F9

Access: RW

Figure 39-4 YCCM_BASE Register



If the Y memory is configured, this register indicates the memory region where the Y memory is mapped in memory. If the Y memory is not configured, this register does not exist. The Y base address needs to be aligned to the size of a memory region and to the size of the memory itself. The Y memory can be mapped to one of the memory regions from 8 to 15.

- If the address bus width is 16, the fields [15:12] indicate the YCCM memory region.
- If the address bus width is 20, the fields [19:16] indicate the YCCM memory region.
- If the address bus width is 24, the fields [23:20] indicate the YCCM memory region.
- If the address bus width is 32, the fields [31:28] indicate the YCCM memory region.

39.3.3 XY Build Configuration Register, XY_BUILD

Address: 0x79

Access: R

Figure 39-5 XY_BUILD Register

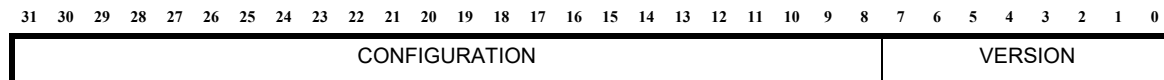


Table 39-2 XY_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of the XY unit 0x20: Current version
CONFIGURATION	[9:8]	XY memory configuration ■ 0: One physical memory: DCCM. ■ 1: Two physical memories: one for DCCM+X and one for Y. ■ 2: Three physical memories: one each for DCCM, X, and Y.
	[10]	Indicates if the XY memory supports interleaving ■ 1: XY memories support interleaving; the X and Y memories are split into even and odd instances to enable single-cycle unaligned accesses. ■ 0: XY memories do not support interleaving.
	[11]	Reserved
	[15:12]	XY memory size ■ 0x0: 4 KB ■ 0x1: 8 KB ■ 0x2: 16 KB ■ 0x3: 32 KB ■ 0x4: 64 KB Note: For DCCM-only configurations (XY_BUILD[9:8]==0), this field is read as zero. In DCCM-only configuration, the XY memory size is the same as the DCCM size, given by DCCM_BUILD[11:8] bits.
	[31:16]	Reserved

40

Address Generation Unit Auxiliary Interface

Address generation can be useful for FFT transforms and FIR delay line implementations. The FFT transforms require bit-reverse carry propagation; whereas, the FIR delay-line implementation requires a module wrapping buffer.

40.1 AGU Auxiliary Register Set

Table 40-1 AGU Registers

Address	Name	Access	Description
0xCC	AGU Build Configuration Register, AGU_BUILD	R	AGU Build Configuration Register
0x5C0 to 0x5CB	AGU Address Pointer Registers, AGU_AUX_APx	rw	Address generation unit address pointer registers
0x5D0 to 0x5D7	AGU Offset Registers, AGU_AUX_OSx	rw	Address generation unit offset registers
0x5E0 to 0x5F7	AGU Modifier Registers, AGU_AUX_MODx	rw	Address generation unit modifier registers

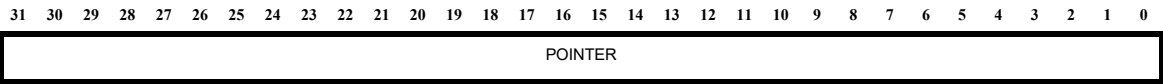
40.1.1 AGU Address Pointer Registers, AGU_AUX_APx

Address: 0x5C0 to 0x5CB

Access: rw

Reset: 0x0000_0000

Figure 40-1 AGU_AUX_APx Register



These registers are used as byte address pointers to the XY or DCCM memories. The number of address pointer registers included in the processor is based on the AGU configuration. The width of the register depends on the ADDR_SIZE configuration option. If ADDR_SIZE < 32, the most significant bits [31: (32-ADDR_SIZE)] are read as zero.

Table 40-2 Number of Address Pointer Registers

AGU Configuration (AGU_SIZE)	Number of Address Pointer Registers
small	4
medium	8
large	12

When the AGU generates an address, it refers to an address-pointer register using the AGU_AUX_MODx.[3:0] field.

After a memory access, the next memory address is calculated by post-incrementing an address pointer in the following ways:

- Increment the address pointer using the offset register value or the offset immediate value.
- Bit-reverse increment of the address pointer using the offset register value or the offset immediate value. Bit-reversing involves the following steps:
 - a. Reverse the bits, that is swap the MSB bits for the LSB bits, in the address pointer register and the offset register.
 - b. Increment the address pointer using the offset value.
 - c. Reverse the bits again in the address pointer register.
- Modulo increment and wrapping: An increment region is created in the XY memory or DCCM with the following boundaries:
 - lower boundary==base address of the memory region

- upper boundary==lower boundary+wrap value

When the address pointer is incremented with the offset (register or immediate), it has to remain in the increment region. If the address pointer post increment crosses the upper boundary, the address pointer wraps to the lower boundary. If the address pointer post increment crosses the lower boundary, the address pointer wraps to the upper boundary.



Note

When you use an AGU register both as a source and a destination operand, the modifier register is updated only once.

For a conditional instruction, the instruction is executed only if the condition is true.

If the condition is false, the instruction is skipped, the address-pointer registers are not updated, are no XY memory loads occur; both the source and destination addresses retain their values from before the update.

Table 40-3 Post-Incrementing Address Pointer Modes

AGU_AUX_MODx.[31:29] Value	Description
0	Linear increment with the signed offset from the register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field.
1	Linear increment with the unsigned immediate offset; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field.
2	Bit reverse increment with offset from register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field.
3	Bit reverse increment with immediate offset; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field.
4	Modulo wrapping increment with offset from register and modulo boundary from register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the wrap register to use is specified in the AGU_AUX_MODx.[17:14] field; and the direction to increment is specified in the AGU_AUX_MODx.[11] field.
5	Power of 2 modulo wrapping increment with immediate offset and immediate power of 2 modulo boundary; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the wrap immediate power is specified in the AGU_AUX_MODx.[18:14] field; and the direction to increment is specified in the AGU_AUX_MODx.[11] field.

Table 40-3 Post-Incrementing Address Pointer Modes (Continued)

AGU_AUX_MODx.[31:29] Value	Description
6	Modulo wrapping increment with offset from register and immediate power of 2 modulo boundary; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the wrap immediate power is specified in the AGU_AUX_MODx.[18:14] field; and the direction to increment is specified in the AGU_AUX_MODx.[11] field.
7	- When AGU_AUX_MODx.SUB_OP==0, update the modifier specified by the SEC_PTR_REG with the address from the AGU_AUX_MODx.[3:0] field and the AGU_AUX_MODx.[28:25] field. For example: $xy\_ptr = AGU\_AUX\_AP[ptr\_reg]$ $AGU\_AUX\_AP[sec\_ptr\_reg] = xy\_ptr + AGU\_AUX\_OS[offset\_reg]$ 7: When AGU_AUX_MODx.SUB_OP==1, update the modifier specified by the SEC_PTR_REG with the address from the AGU_AUX_MODx.[3:0] register and the immediate offset specified by the AGU_AUX_MODx.[21:18] field. For example: $xy\_ptr = AGU\_AUX\_AP[ptr\_reg]$ $AGU\_AUX\_AP[sec\_ptr\_reg] = xy\_ptr + offset\_imm$

**Note**

- The address-pointer value is updated only if the offset is non-zero or if AGU_AUX_MODx.[31:29]==7.
- If multiple instruction operands refer to the same address pointer, the zero offset update is ignored unless AGU_AUX_MODx.[31:29]==7.
- If multiple instruction operands refer to the same address pointer using non-zero offset values or with AGU_AUX_MODx.[31:29]==7, the behavior is undefined.

If insufficient modifiers are available in the hardware configuration, you can use the MODIF instruction explicitly to update the address pointer.

Table 40-4 AGU_AUX_APx Field Description

Field	Bit	Description
POINTER	[31:0]	Byte address pointers to values in the XY or DCCM memories

Example 40-1 Updating the Address Pointer Value

```

// pseudocode for updating the address pointer value
void update_addr(bit_vector<5> mod_reg) { // the modifier index
    mod_t mod; // modifier value (struct as per AUX reg definition)
    mod = AGU_AUX_MOD[mod_reg];
    apply_modifier(mod);
}

int32 bitrev(int32 a) {
    // reverse bit order in an address vector
    int32 r = 0;
    for (int i = 0; i < ADDR_WIDTH; i++) { // ADDR_WIDTH = unumber of bits in address
        r = (r << 1) | (a & 1);
        a >>= 1;
    }
    return r;
}

uint32 log2up(uint32 a) {
    // find next power of 2 greater than or equal to a
    uint32 r = 0;
    while ((1 << r) < a) r++;
    return r;
}

void apply_modifier(mod_t mod) { // the modifier
    uint32 addr; // current address
    uint32 addr_nxt; // next address
    int32 offset; // signed offset
    int32 wrap; // unsigned wrap boundary, minus 1
    int32 mask; // mask for wrapping
    // retrieve pointer value
    addr = AGU_AUX_AP[mod.ptr_reg];
    // get offset
    if ((mod.opc % 2) == 0) {
        offset = AGU_AUX_OS[mod.offset_reg];
    } else {
        offset = unsigned(mod.offset_imm);
    }
}

```

```

// get wrap boundary or scale offset
if (mod.opc == 4) {
    wrap = AGU_AUX_OS[mod.wrap_reg];
} else if (mod.opc == 5 || mod.opc == 6) {
    wrap = unsigned(power(2, mod.wrap_imm+1))-1;
} else {
    offset = offset << mod.sc;
}
// flip pointer and offset bits if bit-reverse
if (mod.opc == 2 || mod.opc == 3) {
    addr = bitrev(addr);
    offset = bitrev(offset);
}
// optionally negate offset
if (mod.dir)
    offset = -offset;
addr_nxt = addr + offset;
if (mod.opc == 2 || mod.opc == 3) {
    // flip back pointer bits if bit-reverse, avoid carry prop into lsbs
    addr_nxt = bitrev(addr_nxt); // bitrev is flipping bits in 32b word
    // create a mask to clear lsbs
    if (mod.repl)
        lsbmask = -power(2, mod.ds);
    else
        lsbmask = -power(2, mod.vw-mod.cs+mod.ds);
    addr_nxt &= lsbmask;
} else if (mod.opc >= 4) {
    // wrap
    mask = power(2, log2up(wrap+1))-1; // log2up is base 2 logarithm rounded up
    if (((addr_nxt & ~mask) != (addr & ~mask)) || ((addr_nxt & mask) > wrap)) {
        // went through the wrap boundary
        if (offset > 0) {
            addr_nxt -= wrap + 1; // went over max
        } else {
            addr_nxt += wrap + 1; // went under min
        }
    }
} // else transparent
// update address with next value
AGU_AUX_AP[mod.ptr_reg] = addr_nxt;
}

```

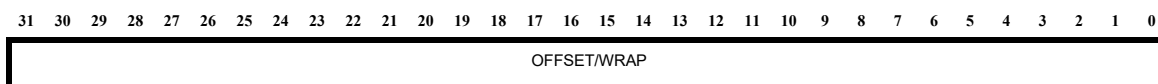
40.1.2 AGU Offset Registers, AGU_AUX_OSx

Address: 0x5D0 to 0x5D7

Access: rw

Reset: 0x0000_0000

Figure 40-2 AGU_AUX_OSx Register



These registers are used to determine the increment values and wrap boundaries during the address generation process. If used as an offset value, the value is the non-scaled value that is added to or subtracted from the address pointer. If used as a wrap boundary, this value should specify the size of the modulo buffer in bytes minus one.

The width of the register depends on the ADDR_SIZE configuration option. If ADDR_SIZE < 32, the most significant bits [31: (32-ADDR_SIZE)] are read as zero.

The number of offset or wrap registers included in the processor is based on the AGU configuration.

Table 40-5 Number of Offset or Wrap Registers

AGU Configuration (AGU_SIZE)	Number of Offset or Wrap Registers
small	2
medium	4
large	8

After a memory access, the pointer value in the address pointer register is incremented based on the rules in [Table 40-3](#) on page [1203](#).

40.1.3 AGU Modifier Registers, AGU_AUX_MODx

Address: 0x5E0 to 0x5F7

Access: rw

Reset: 0x0000_0000

The modifier registers define the type of address arithmetic to be performed during address generation and also the data transformations performed on the data vectors. The modifier registers allow you to create data structures in memory. The format of the modifier registers depends on the addressing mode and the data transformation to be performed.

Table 40-6 Number of Modifier Registers

AGU Configuration (AGU_SIZE)	Number of Modifier Registers
small	4 (addresses 0x5E0 to 0x5E3 are used within the AGU modified register range (0x5E to 0x5F7, the unused registers are reserved).
medium	12
large	24

Figure 40-3 AGU_AUX_MODx Register with Offset Register Increment

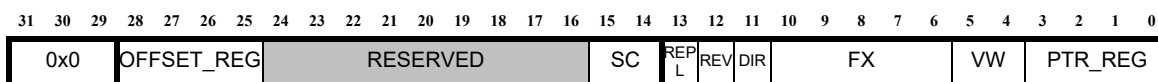


Figure 40-4 AGU_AUX_MODx Register with Immediate Offset

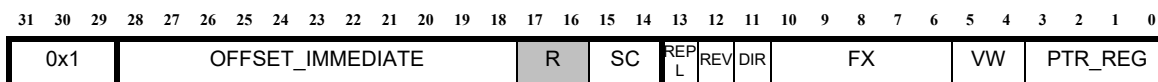


Figure 40-5 AGU_AUX_MODx Register with Bit Reverse Increment Using Offset Register

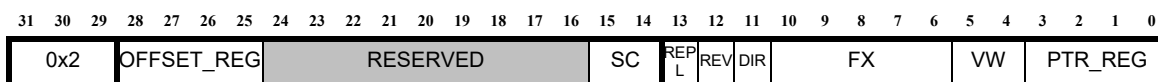


Figure 40-6 AGU_AUX_MODx Register with Bit Reverse Increment Using Immediate Offset

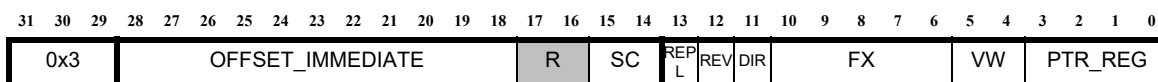
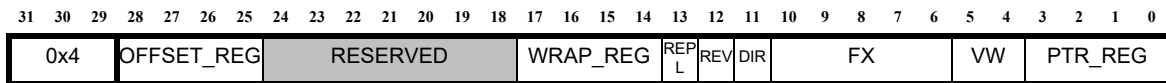
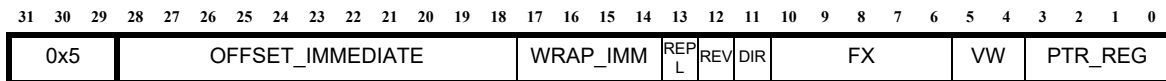
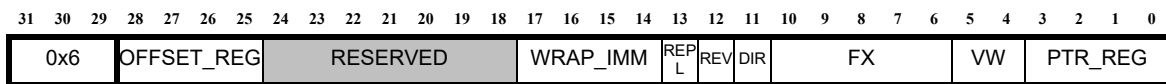
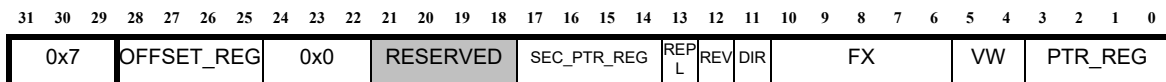
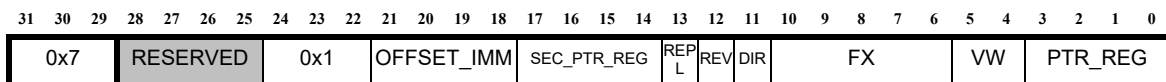


Figure 40-7 AGU_AUX_MODx Register with Modulo Wrapping Using Offset and Wrap Registers**Figure 40-8 AGU_AUX_MODx Register with Modulo Wrapping Using Immediate Offset and Wrap****Figure 40-9 AGU_AUX_MODx with Modulo Wrapping Using Offset Register and Immediate Wrap****Figure 40-10 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset Register****Figure 40-11 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset Immediate**

Field	Bit	Description
PTR_REG	[3:0]	Indicates the address pointer register index
DIR	11	Indicates if the address pointer should be incremented or decremented. 1: decrement the address pointer 0: increment the address pointer
SC	[15:14]	Indicates the offset scaling factor. $\text{effective\_offset} = \text{offset} * 2^{\text{SC}}$
WRAP_IMM	[17:14]	Specifies the unsigned immediate wrap value; this value is used to create a region within the XY memory with the following boundaries: - lower_boundary == base address of the XY memory region - upper_boundary == lower_boundary + $2^{\text{WRAP\_IMM}+1}$ The address pointer has to remain within this region. If the address pointer goes out of bounds of this region, the address pointer wraps to the lower or the upper boundary of the region. This field is available as a wrap immediate only if $\text{OPC} == 5 \mid 6$.
WRAP_REG	[17:14]	Specifies the index of the wrap register that holds the unsigned wrap value. This field is available only if $\text{OPC} == 4$. $\text{upper\_boundary} == \text{lower\_boundary} + \text{offset}[\text{WRAP\_REG}]$

Field	Bit	Description
SEC_PTR_REG	[17:14]	Specifies a destination AGU Address Pointer Register (AGU_AUX_APx) defining the address pointer to be used for the address update
SUB_OP	[24:22]	This field is specific to <code>OPC==7</code> . Specifies whether an offset register or offset immediate is used in the pointer arithmetic.
OFFSET_IMM	[28:18]	Specifies the unsigned immediate offset value by which the address pointer is incremented or decremented. This field is available only if <code>OPC==1 3 6 7</code> . When <code>OPC=7</code> , the <code>OFFSET_IMM</code> field is at position [21:19], the bits [24:22] are taken up by the <code>SUB_OP</code> field, and [28:25] are reserved.
OFFSET_REG	[28:25]	Specifies the index of the offset register that holds the signed offset value. This field is available only if <code>OPC==0 2 4 5 7</code> .

Field	Bit	Description
OPC	[31:29]	This field controls the arithmetic to increment or decrement the address pointer. 0: Linear increment with signed offset from register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field. 1: Linear increment with unsigned immediate offset; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field. 2: Bit reverse increment with offset from register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field. 3: Bit reverse increment with immediate offset; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field. 4: Modulo wrapping increment with offset from register and modulo boundary from register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the wrap register to use is specified in the AGU_AUX_MODx.[17:14] field; and the direction to increment is specified in the AGU_AUX_MODx.[11] field. 5: Power of 2 modulo wrapping increment with immediate offset and immediate power of 2 modulo boundary; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the wrap immediate power is specified in the AGU_AUX_MODx.[18:14] field; and the direction to increment is specified in the AGU_AUX_MODx.[11] field. 6: Modulo wrapping increment with offset from register and immediate power of 2 modulo boundary; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the wrap immediate power is specified in the AGU_AUX_MODx.[18:14] field; and the direction to increment is specified in the AGU_AUX_MODx.[11] field. 7: When SUB_OP==0, update the modifier specified by the SEC_PTR_REG with the address from the AGU_AUX_MODx.[3:0] field and the AGU_AUX_MODx.[28:25] field. For example: <pre>xy_ptr = AGU_AUX_AP[ptr_reg] AGU_AUX_AP[sec_ptr_reg] = xy_ptr + AGU_AUX_OS[offset_reg]</pre> 7: When SUB_OP==1, update the modifier specified by the SEC_PTR_REG with the address from the AGU_AUX_MODx.[3:0] register and the immediate offset specified by the AGU_AUX_MODx.[21:18] field. For example: <pre>xy_ptr = AGU_AUX_AP[ptr_reg] AGU_AUX_AP[sec_ptr_reg] = xy_ptr + offset_imm</pre> Note: For a description of the bit-reverse increment and modulo wrapping increment methods, see “Address Generation” on page 1192.

Data Transformation

The modifier registers also let you perform data transformation on the information in the XY memory. The information in the XY memory is packaged as follows:

- Data: information coming from the memory
- Container: Expanded data. Data may be expanded into a wider container
- Vector: One or multiple containers

Table 40-7 lists the modifier register fields that define the data transformations you can perform:

Table 40-7 Data Transformation-Related Fields in the Modifier Registers

Field	Bit	Description
VW	[5:4]	Vector width in bytes. Note: This value must not be less than the container size.
FX	[10:6]	Data and container packing information for the source operands; See Table 40-8. Note: the FX values 15 and 23 to 31 are undefined.
REV	12	Reverse the container ordering in the vector
REPL	13	Replicate the first container across the full width of the vector

Table 40-8 Data Transformation Modes Based on the FX Field

Fx	Container Size in Bits	Data Size in Bits	Expand and Align
0	16	8	LSB-aligned expansion with sign extension
1	16	8	LSB-aligned extend with zero
2	16	8	MSB-aligned expansion with zero in the LSB bits
3	16	16	The container is the same size as the data
4	32	8	LSB-aligned expansion with sign extension
5	32	8	LSB-aligned extend with zero
6	32	8	MSB-aligned expansion with zero in the LSB bits
7	8	8	The container is the same size as the data
8	32	16	LSB-aligned expansion with sign extension
9	32	16	LSB-aligned extend with zero
10	32	16	MSB-aligned expansion with zero in the LSB bits

Table 40-8 Data Transformation Modes Based on the FX Field

Fx	Container Size in Bits	Data Size in Bits	Expand and Align
11	32	32	The container is the same size as the data
12	64	8	LSB-aligned expansion with sign extension
13	64	8	LSB-aligned extend with zero
14	64	8	MSB-aligned expansion with zero in the least-significant bits
16	64	16	LSB-aligned expansion with sign extension
17	64	16	LSB-aligned extend with zero
18	64	16	MSB-aligned expansion with zero in the least-significant bits
19	64	64	The container is the same size as the data
20	64	32	LSB-aligned expansion with sign extension
21	64	32	LSB-aligned extend with zero
22	64	32	MSB-aligned expansion with zero in the LSB bits

You can use the modifier registers to perform the following data transformations on the source operand modifiers:

- Resize the data from XY memory while zero-extending or sign-extending the data
- Replicate the scalars from XY memory across the width of a vector
- Reverse the order of the elements of a vector

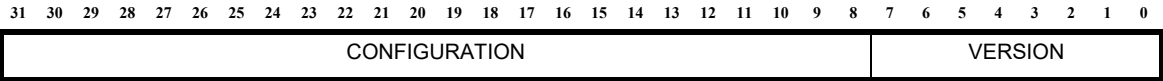
The destination operands support the following data transformations:

- Limit the width of the write-back operation
- Swap data in containers (for example, endian conversion)
- Reverse vector container order (for example, endian conversion)
- Replicate a container across a vector YCCM_BASE

40.1.4 AGU Build Configuration Register, AGU_BUILD

Address: 0xCC
Access: R

Figure 40-12 AGU_BUILD Register



This register indicates the presence of the Address Generator Unit (AGU) and its configuration.

Table 40-9 AGU_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of the AGU unit 0x01: first version 0x02: current version; supports copy and arithmetic on the AGU address pointers.
CONFIGURATION	[11:8]	Number of address pointers The number of address pointers is defined by the build option <code>agu_size</code> . ■ If <code>agu_size==small</code>; number of address pointers = 4 ■ If <code>agu_size==medium</code>; number of address pointers = 8 ■ If <code>agu_size==large</code>; number of address pointers = 12
	[15:12]	Number of offset registers The number of offset registers is defined by the build option <code>agu_size</code> . ■ If <code>agu_size==small</code>; number of offset registers = 2 ■ If <code>agu_size==medium</code>; number of offset registers = 4 ■ If <code>agu_size==large</code>; number of offset registers = 8
	[20:16]	Number of modifiers The number of modifier registers is defined by the build option <code>agu_size</code> . ■ If <code>agu_size==small</code>; number of modifier registers = 4 ■ If <code>agu_size==medium</code>; number of modifier registers = 12 ■ If <code>agu_size==large</code>; number of modifier registers = 24
	[23:21]	Write buffer size ■ 0x2: the buffer can store 2 x 32 bits ■ 0x4: the buffer can store 4 x 32 bits
	[24]	Indicates the presence of the accordion pipeline stage ■ 1: the accordion pipeline stage is included; the XY-logic is included as a separate stage in the pipeline. ■ 0: the accordion pipeline stage is not included; the XY-logic becomes part of the fetch stage in the pipeline.
	[31:26]	Reserved

41

Bitstream Processing

Use the auxiliary registers in [Table 41-1](#) to program and control bitstreams.

Table 41-1 Bitstream Auxiliary Registers

Address	Name	Access	Description
0x5FC	Bitstream Control Register, BS_AUX_CTRL	rw	Bitstream control register.
0x5FD	Bitstream Base Address Register, BS_AUX_ADDR	rw	Bitstream base address register
0x5FE	Bitstream Offset Register, BS_AUX_BIT_OS	rw	Bitstream offset register
0x5FF	Bitstream Write Data Register, BS_AUX_WDATA	rw	Bitstream data register

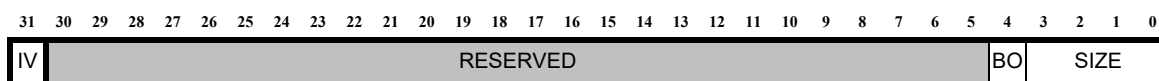
41.0.1 Bitstream Control Register, BS_AUX_CTRL

Address: 0x5FC

Access: rw

Reset: 0x00

Figure 41-1 BS_AUX_CTRL Register



Use this register to control the bitstream features such as the bitstream buffer size, bit ordering, and whether to invalidate prefetched data or not.

Table 41-2 BS_AUX_CTRL Field Description

Field	Bit	Description
SIZE	[3:0]	Specifies the size of the XY memory buffer in KB Buffer size = 2^{SIZE} KB.
BO	4	Specifies the bit ordering in a byte. 1: Bitstream starts at the MSB. 0: Bitstream starts at the LSB.
IV	31	Invalidates the prefetched data 1: Prefetched data is invalidated and the bit offset register is set to 0 When read, this bit always reads 0.

Example 41-1 BS_AUX_CTRL.BO Bit Behavior

```
mem[0] = 0xCA
mem[1] = 0xFE
mem[2] = 0xAB
mem[3] = 0xBA
bspeek(8) // returns 0xCA
bspeek(16) // returns 0xCAFE if BS_AUX_CTRL.BO is true else 0xFECA if false
bspeek(2) // returns 0x3 if BS_AUX_CTRL.BO is true else 0x2 if false
```

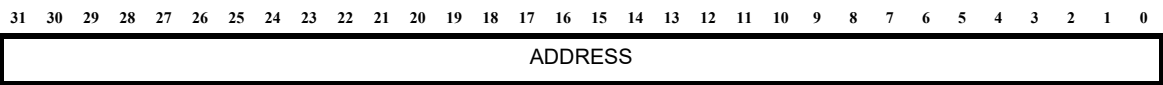
41.0.2 Bitstream Base Address Register, BS_AUX_ADDR

Address: 0x5FD

Access: rw

Reset: 0x00

Figure 41-2 BS_AUX_ADDR Register



Use this register to specify the bitstream base address, as a byte address, in the XCCM, YCCM, or DCCM memories. The base address should be aligned to the buffer size (BS_AUX_CTRL.SIZE). For example, if the buffer size is 2 KB, the base address should be aligned at a 2 KB address boundary.

Table 41-3 BS_AUX_ADDR Field Description

Field	Bit	Description
ADDRESS	[31:0]	Specifies the size of the bitstream buffer in the XCCM, YCCM, or DCCM memories.

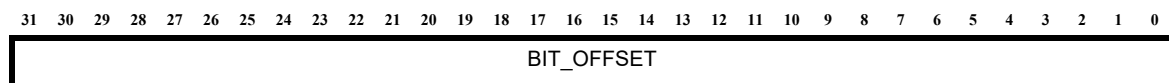
41.0.3 Bitstream Offset Register, BS_AUX_BIT_OS

Address: 0x5FE

Access: rw

Reset: 0x00

Figure 41-3 BS_AUX_BIT_OS Register



Use this register to specify the bit offset from the bitstream base address, BS_AUX_ADDR. The actual byte address accessed in the bitstream buffer is $BS\_AUX\_ADDR + (BS\_AUX\_BIT\_OS / 8)$. This register is reset to zero when the prefetched data is invalidated, BS_AUX_CTRL.IV==1.

Table 41-4 BS_AUX_BIT_OS Field Description

Field	Bit	Description
BIT_OFFSET	[31:0]	Specifies the bit offset from the bitstream base address.

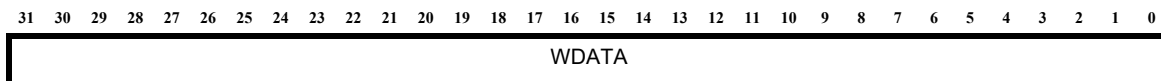
41.0.4 Bitstream Write Data Register, BS_AUX_WDATA

Address: 0x5FF

Access: rw

Reset: 0x00

Figure 41-4 BS_AUX_WDATA Register



Use this register to store partial write data for pushing data to the bitstream buffer using the BSPUSH instruction. The data from this register is flushed to the bitstream buffer only after the register is full, that is, 32 bits have been pushed. This register can be read or written using SR/LR instructions without triggering a write-back.

Table 41-5 BS_AUX_WDATA Field Description

Field	Bit	Description
WDATA	[31:0]	Stores the partial write data for bitstream operations.

41.1 Build Configuration Registers

The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

A set of baseline Build Configuration Registers are present in all ARCV2 processors. In addition, each optional ISA extension typically includes a Build Configuration Register to signify its presence and its specific configuration in each given build.

Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCV2-based system.

Auxiliary registers, in the range 0x60 to 0x7F and 0xC0 to 0xFF, are assumed to be BCRs. In kernel mode, any read from a non-existent build configuration register in this range returns 0, and no exception is generated. This design enables the kernel-mode code to detect the presence or absence of a BCR because all BCRs that are present in a system contain non-zero values.

Table 41-6 Build Configuration Registers

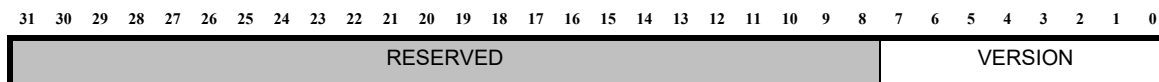
Number	Name	Access	Description
0xCB	Bitstream Identity Register, BS_BUILD	R	Build configuration register: bitstream interface

41.1.1 Bitstream Identity Register, BS_BUILD

Address: 0xCB

Access: R

Figure 41-5 BS_BUILD Register



This register indicates the presence of the bitstream interface and its configuration. This register is present in the build only if `has_bs==true`.

Table 41-7 BS_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of the bitstream interface 0x01: Initial version 0x02: Current version

41.2 Exceptions

See [Table 6-6](#) on page 268.

42

User Auxiliary Register Interface

When a user auxiliary register interface is included in the processor build configuration, the upper auxiliary address range (0x8000_0000 to 0xFFFF_FFFF) is allocated to the external user auxiliary registers.

The user auxiliary registers can be assigned access based on the [“Key to Auxiliary Register Access Permissions for LR and SR Instructions”](#).

42.1 UAUX in Secure Mode

The signal `uaux_s_mode` indicates to the core that the secure access mode of a UAUX register. If this signal is 1, it indicates the selected register can only be accessed from the secure mode, any access from the normal mode causes privilege violation (ECR = 0x071020). If this signal is 0, it indicates the selected register can be accessed from both the secure and normal modes. The user logic must implement the secure mode for the UAUX registers. Ensure that the user logic for the secure mode cannot be altered during run-time.

You can also protect UAUX registers using SID protection. The signal `uaux_sid` indicates to the core that the SID assigned to each UAUX register. When `mpu_sid_option=true`, on access to a UAUX register, the SID of that UAUX register is compared with the SID of the instruction (generated by MPU). If the bit-wise comparing result is 0, it indicates an SID mismatch and causes privilege violation (ECR = 0x071040). The SID of each UAUX register is defined by user logic and the SID can be dynamically programmed.

If both secure mode violation and SID violation occur together, the secure mode violation takes precedence and the ECR 0x071020 is generated.

43

ARConnect

43.1 ARConnect Introduction

ARConnect is a collection of system components that provides efficient inter-core communications, interrupt distribution, debug assistance, and other inter-core functionality in a multi-core system. Use ARConnect to integrate multiple ARC cores into multi-core systems that can be used to run applications such as the divide-and-conquer parallel applications or pipeline stream applications.

43.2 ARConnect Register Interface

Table 43-1 ARConnect Registers

Address	Name	Description
0X600	CONNECT_CMD	ARConnect command register. Each core uses this register to send commands to the ARConnect. This register is present only when ARConnect is configured.
0x601	CONNECT_WDATA	Write data register. Each core uses this register to write data to ARConnect. This register is present only when ARConnect is configured.
0x602	CONNECT_READBACK	Read data register. Each core uses this register to read data from ARConnect. This register is present only when ARConnect is configured.
0xD0	CONNECT_SYSTEM_BUILD	Build configuration for: ARConnect System
0xD1	CONNECT_SEMAPHORE_BUILD	Build configuration for: ARConnect inter-core semaphore unit
0xD2	CONNECT_MESSAGE_BUILD	Build configuration for: ARConnect inter-core message unit
0xD3	CONNECT_PMU_BUILD	Build configuration for: ARConnect power management unit
0xD6	CONNECT_GFRC_BUILD	Build configuration for: ARConnect global free running counter

Table 43-1 (Continued)ARConnect Registers

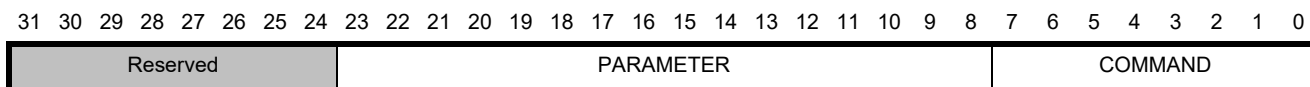
Address	Name	Description
0xE0	CONNECT_ICI_BUILD	Build configuration for: ARConnect inter-core interrupt unit
0xE1	CONNECT_ICD_BUILD	Build configuration for: ARConnect inter-core debug unit
0xE3	CONNECT_PDM_BUILD	Build configuration for: ARConnect power domain management unit

43.2.1 ARConnect Command Register, `CONNECT_CMD`

Address: 0x600

Access: RW

Figure 43-1 `CONNECT_CMD` Register



The `CONNECT_CMD` register is used to send commands to ARConnect. For example, an ARC core can use the SR instruction to write specific commands to the `CONNECT_CMD` register, and the ARC core passes this command to ARConnect (to generate interrupts to other cores, or pass messages to other cores, and so on). The ARC core can use the LR instruction to read the `CONNECT_CMD` register to check the content of the last written command. In earlier releases, this register was named `MCIP_CMD`.

Table 43-2 `CONNECT_CMD` Register Field Description

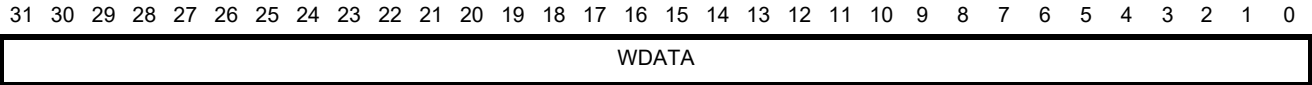
Field	Bits	Reset	Description
COMMAND	[7:0]	0	Command to ARConnect.
PARAMETER	[23:8]	0	Parameter field of the command to ARConnect. Any parameter that is required by a command is supplied in this field.
RESERVED	[31:24]	0	Reserved. Read-as-zero; ignored on write

43.2.2 ARConnect Write Data Register, CONNECT_WDATA

Address: 0x601

Access: RW

Figure 43-2 CONNECT_WDATA Register



The CONNECT_WDATA register is used to store data that should be passed from the ARC core to ARConnect. Some ARConnect commands use the data that you specify in this register.

The layout of the CONNECT_WDATA register depends on the ARConnect command being issued. In earlier releases, this register was named MCIP_WDATA.

Table 43-3 CONNECT_WDATA Register Field Description

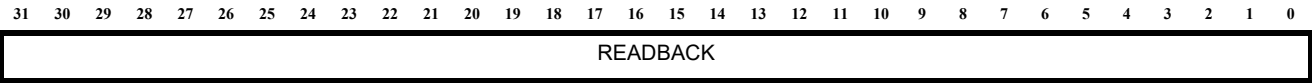
Field	Bits	Reset	Description
WDATA	[31:0]	0	The store data passed from ARC core to ARConnect

43.2.3 ARConnect Read Data Register, CONNECT_READBACK

Address: 0x602

Access: R

Figure 43-3 CONNECT_READBACK Register



The CONNECT_READBACK register returns the read back data when an ARC core issues any ARConnect-related read command. This register is read-only; writes to it are ignored and an illegal-instruction exception is raised.

The layout of the CONNECT_READBACK register depends on the issued ARConnect command. In earlier releases, this register was named MCIP_READBACK.

Table 43-4 CONNECT_READBACK Register Field Description

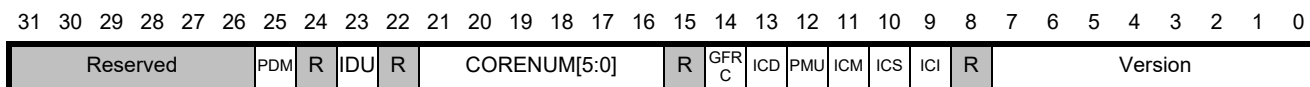
Field	Bits	Reset	Description
READBACK	[31:0]	0	The read back data when an ARC core issues any ARConnect related read command

43.2.4 ARConnect Build Configuration Register, CONNECT_SYSTEM_BUILD

Address: 0xD0

Access: R

Figure 43-4 CONNECT_SYSTEM_BUILD Register



The `CONNECT_SYSTEM_BUILD` register is an optional BCR register in the ARC cores. If the ARC core is not configured with ARConnect, reading this register returns zero. In earlier releases, this register was named as `MCIP_SYSTEM_BUILD`.

Table 43-5 CONNECT_SYSTEM_BUILD Register Field Description

Field	Bits	Description
VERSION	[7:0]	The ARConnect version. All non-enumerated values are reserved. 0x1: First version 0x2: Current version; - Changed the starting <code>CORE_ID</code> to 0 from 1 (first version). - Renamed the component from <code>MCIP</code> to <code>ARCONNECT</code>.
ASI	[8]	Presence of ARConnect Slave interface Unit 0x0: No ARConnect slave unit 0x1: Has ARConnect slave unit
ICI	[9]	Presence of Inter-Core Interrupt Unit 0x0: No inter-core interrupt unit 0x1: Has inter-core interrupt unit
ICS	[10]	Presence of Inter-Core Semaphore Unit 0x0: No inter-core semaphore unit 0x1: Has inter-core semaphore unit
ICM	[11]	Presence of Inter-Core Message Unit 0x0: No inter-core message unit 0x1: Has inter-core message unit
PMU	[12]	Presence of Power Management Unit 0x0: No power management unit 0x1: Has power management unit

Table 43-5 CONNECT_SYSTEM_BUILD Register Field Description

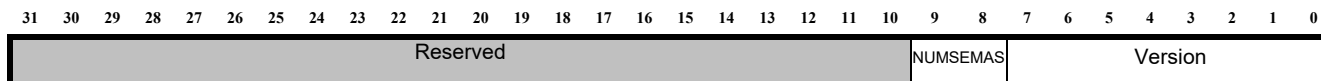
Field	Bits	Description
ICD	[13]	Presence of Inter-Core Debug Unit 0x0: No inter-core debug unit 0x1: Has inter-core debug unit
GFRC	[14]	Presence of Global Free-Running Counter Unit 0x0: No global free-running counter 0x1: Has global free-running counter
RESERVED	[15]	Reserved. Read-as-zero; ignored on write
CORENUM_5_0_	[21:16]	Number of cores (ARC and non-ARC) connected to ARConnect For example 6'h01: 1 core; 6'h02: 2 cores; 6'h04: 4 cores.
RESERVED	[22]	Reserved. Read-as-zero; ignored on write
IDU	[23]	Presence of Interrupt Distribution Unit 0x0: No interrupt distribution unit 0x1: Has interrupt distribution unit
RESERVED	[24]	Reserved. Read-as-zero; ignored on write
PDM	[25]	Presence of Power Domain Management Unit 0x0: No ARConnect power domain management unit 0x1: Has ARConnect power domain management unit
RESERVED	[31:27]	Reserved. Read-as-zero; ignored on write

43.2.5 Inter-core Semaphore Unit BCR, CONNECT_SEMA_BUILD

Address: 0xD1

Access: R

Figure 43-5 CONNECT_SEMA_BUILD Register



The `CONNECT_SEMA_BUILD` register is an optional BCR register. If the Inter-Core Semaphore Unit is not configured in ARConnect, reading the `CONNECT_SEMA_BUILD` register returns zero. In earlier releases, this register was named `MCIP_SEMA_BUILD`.

Table 43-6 CONNECT_SEMA_BUILD Register Field Description

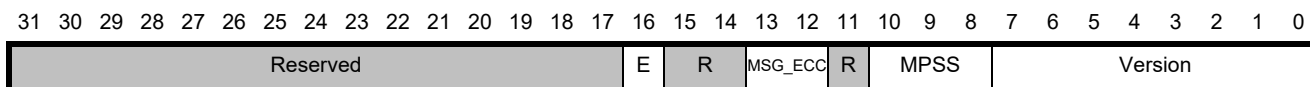
Field	Bits	Description
Version	[7:0]	Inter-core Semaphore Unit version. 0x1: Current version
NUMSEMAS	[9:8]	Number of Semaphores: 0x0: 8 semaphores 0x1: 16 semaphores 0x2: 32 semaphores 0x3: reserved
RESERVED	[31:10]	Reserved. Read-as-zero; ignored on write

43.2.6 Inter-core Message Unit BCR, CONNECT_MESSAGE_BUILD

Address: 0xD2

Access: R

Figure 43-6 CONNECT_MESSAGE_BUILD Register



The `CONNECT_MESSAGE_BUILD` register is an optional BCR register. If the Inter-Core Message Unit is not configured in ARConnect, reading the `CONNECT_MESSAGE_BUILD` register returns zero. In earlier releases, this register was named `MCIP_MESSAGE_BUILD`. This register exists only when `mcip_has_msg_sram==true` in your processor build.

Table 43-7 CONNECT_MESSAGE_BUILD Register Field Description

Field	Bits	Description
Version	[7:0]	Inter-core Message Unit version. 0x1: initial version. 0x2: Support for ECC protection on SRAM. 0x3: Added single-bit error counter for ECC.
MPSS	[10:8]	Size of the message-passing SRAM 0x0: 128 B 0x1: 256 B 0x2: 512 B 0x3: 1 KB 0x4: 2 KB 0x5: 4 KB - Other values are reserved
RESERVED	[11]	Reserved. Read-as-zero; ignored on write
MSG_ECC	[13:12]	ECC or parity protection scheme of Message SRAM 0x0: No protection 0x1: Parity protection 0x2: ECC (SECCDED) protection
RESERVED	[15:14]	Reserved. Read-as-zero; ignored on write

Table 43-7 CONNECT_MESSAGE_BUILD Register Field Description

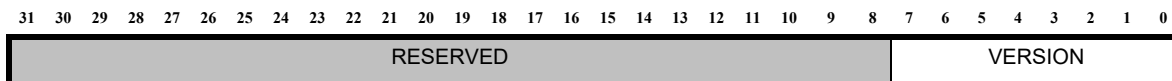
Field	Bits	Description
E (ECC_ADDR)	[16]	Specifies whether ECC protection is configured on message SRAM address ■ 0x0: No protection ■ 0x1: ECC protection is configured
RESERVED	[31:17]	■ Reserved. Read-as-zero; ignored on write

43.2.7 Power Management Unit BCR, CONNECT_PMU_BUILD

Address: 0xD3

Access: R

Figure 43-7 CONNECT_PMU_BUILD Register



The `CONNECT_PMU_BUILD` register is an optional BCR. If the PMU is not configured in ARConnect, reading the `CONNECT_PMU_BUILD` register returns zero. In earlier releases, this register was named `MCIP_PMU_BUILD`.

Table 43-8 CONNECT_PMU_BUILD Field Description

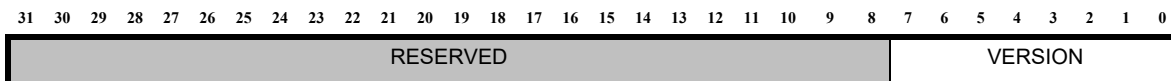
Field	Bit	Description
VERSION	[7:0]	Version number ■ 0x1: First version ■ 0x2: Second version; deleted the DVFS feature from ARConnect PMU; moved the power management feature to PDM (Power Domain Management); redefined PMU as optional external Power Management Unit. ■ 0x3: Current version; Added two additional counters (<code>RSTCNT</code> and <code>PDCNT</code>) and four relevant commands: <code>CMD_PMU_SET_RSTCNT</code>, <code>CMD_PMU_READ_RSTCNT</code>, <code>CMD_PMU_SET_PDCNT</code>, <code>CMD_PMU_READ_PDCNT</code>
RESERVED	[31:27]	■ Reserved. Read-as-zero; ignored on write

43.2.8 ARConnect Global Free Running Counter BCR, CONNECT_GFRC_BUILD

Address: 0xD6

Access: R

Figure 43-8 CONNECT_GFRC_BUILD Register



The `CONNECT_GFRC_BUILD` register is an optional BCR. If the global free running counter is not configured in ARConnect, reading the `CONNECT_GFRC_BUILD` register returns zero.

Table 43-9 CONNECT_GFRC_BUILD Field Description

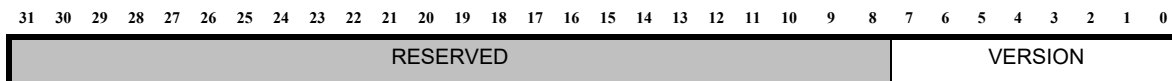
Field	Bit	Description
VERSION	[7:0]	Version number ■ 0x1: First version ■ 0x2: Second version; Renamed the global real-time counter (GRTC) to global free running counter (GFRC). ■ 0x3: Third Version; Added the following commands: <code>CMD_GFRC_SET_CORE</code>, <code>CMD_GFRC_READ_CORE</code>, and <code>CMD_GFRC_READ_HALT</code>. ■ 0x4: Current version; Added the following commands: <code>CMD_GFRC_CLK_ENABLE</code>, <code>CMD_GFRC_CLK_DISABLE</code>, <code>CMD_GFRC_READ_CLK_STATUS</code>
RESERVED	[31:8]	■ Reserved. Read-as-zero; ignored on write

43.2.9 Inter-Core Interrupt Vector Control Unit BCR, CONNECT_IVC_BUILD

Address: 0xD7

Access: R

Figure 43-9 CONNECT_IVC_BUILD Register



The `CONNECT_IVC_BUILD` register is an optional BCR that indicates support for setting the interrupt vector base using an external pin. If the interrupt vector base control unit is not configured in ARConnect, reading the `CONNECT_IVC_BUILD` register returns zero.

Table 43-10 CONNECT_IVC_BUILD Field Description

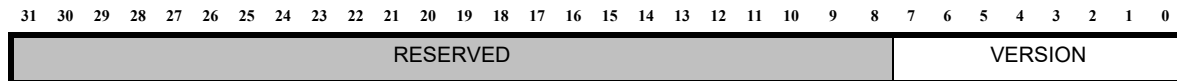
Field	Bit	Description
VERSION	[7:0]	Version number 0x1 : initial version. Indicates support for setting the interrupt vector base using an external pin.
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write

43.2.10 Inter-Core Interrupt Unit BCR, CONNECT_ICI_BUILD

Address: 0xE0

Access: R

Figure 43-10 CONNECT_ICI_BUILD Register



The `CONNECT_ICI_BUILD` register is an optional BCR. If the inter-core interrupt unit is not configured in ARConnect, reading the `CONNECT_ICI_BUILD` register returns zero.

Table 43-11 CONNECT_ICI_BUILD Field Description

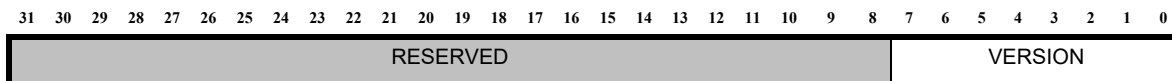
Field	Bit	Description
VERSION	[7:0]	Version number ■ 0x2 : This register did not exist in earlier versions of ARConnect (version 0x1). ■ 0x3: Current version; Added the following ICI command: <code>CMD_INTRPT_GENERATE_ACK_BIT_MASK</code>.
RESERVED	[31:8]	■ Reserved. Read-as-zero; ignored on write

43.2.11 Inter-Core Debug Unit BCR, CONNECT_ICD_BUILD

Address: 0xE1

Access: R

Figure 43-11 CONNECT_ICD_BUILD Register



The `CONNECT_ICD_BUILD` register is an optional BCR. If the inter-core debug unit is not configured in ARConnect, reading the `CONNECT_ICD_BUILD` register returns zero.

Table 43-12 CONNECT_ICD_BUILD Field Description

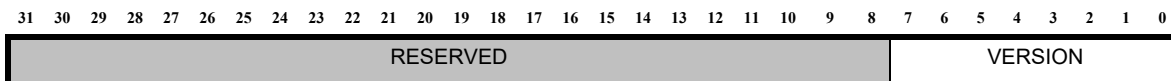
Field	Bit	Description
VERSION	[7:0]	Version number 0x2 : Current version. This register did not exist in earlier versions of ARConnect.
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write

43.2.12 ARConnect PDM BCR, CONNECT_PDM_BUILD

Address: 0xE3

Access: R

Figure 43-12 CONNECT_PDM_BUILD Register



The `CONNECT_PDM_BUILD` register is an optional BCR (exists when `mcip_power_domains==true`). If the power-domain management unit is not configured in ARConnect, reading the `CONNECT_PDM_BUILD` register returns zero.

Table 43-13 CONNECT_PDM_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number 0x1 : Current version
RESERVED	[31:8]	■ Reserved. Read-as-zero; ignored on write

43.3 Programming Restrictions

43.3.1 Atomic Operation

Operation sequences to the ARConnect are considered as atomic operations. The software must save and store the auxiliary registers if the atomic operation is interrupted.

For example: If CORE1 writes the PMU mode, the following instructions must be executed:

- (1) `SR 0x00000001, [CONNECT_WDATA];`
- (2) `SR 0x00000051, [CONNECT_CMD];`

If an interrupt occurs between these two instructions, the sequence is disrupted. The software must save the `CONNECT_WDATA` value, if it is written in the interrupt handler, and restore the value before it exits the interrupt.

43.3.2 Preventing Conflicts

Some ARConnect components such as the PMU require the software to guarantee that no simultaneous-access conflict happens. If two or more cores read or write these components simultaneously, the hardware result is unpredictable.

For example: If CORE1 sets the ICD mode for ICD, it uses the following operation sequence:

1. Read the ICD internal status register
2. Check the internal status

3. Update the ICD internal status register

Meanwhile, if CORE2 sets the IDU mode simultaneously, the operation sequences from two cores could be overlapped. Only one of them can be selected and performed, and the other sequence might be discarded. Therefore, software needs to guarantee no such access conflict happens.

Use the following methods to ensure that there are no simultaneous accesses:

- Utilize the ARConnect Inter-core Semaphores.
- Utilize the ARConnect Inter-core Message Unit to pass specific messages synchronizing different cores.
- Use specific address in the shared memory locked by LLOCK/SCOND as software semaphore if these two instructions are supported.

44

micro DMA Controller

The micro-DMA controller (DMAC) provides a fast and efficient, low energy way of copying large blocks of data in memory, offloading the work from the processor. The DMA controller allows fast DMA transfers with low gate count and low power consumption.

44.1 DMAC Features

- One to 16 independent programmable DMA channels (configurable)
- User-programmable prioritization scheme for all channels
- Concurrent operation with the CPU
- Software and hardware triggered DMA transfers
- Two addressing modes
- One to five data-transfer modes (configurable)
- Internal and external interrupt support

The DMA controller is tightly coupled to the ARC EM processor core interfaces to achieve low latency and cycle efficient DMA transfers optimized to reduce energy consumption.

44.2 DMAC Auxiliary Register Interface

A DMA controller provides a fast and efficient, low energy way of copying large blocks of data in memory, off loading the work from the processor. The DMA controller allows fast DMA transfers, with low gate count and low power consumption. [Table 44-1](#) lists the DMAC auxiliary registers.

Table 44-1 DMAC Auxiliary Registers

Address	Name	Description
0x680	DMA Controller Configuration Register, DMACTRL	DMA controller control register.
0x681	DMA Channel Enable Register, DMACENB	DMA channel enable register

Table 44-1 DMA Auxiliary Registers

Address	Name	Description
0x682	DMA Channel Disable Register, DMACDSB	DMA channel disable register
0x683	DMA Channel High Priority Level Register, DMACHPRI	DMA channel high priority register
0x684	DMA Channel Normal Priority Level Register, DMACNPRI	DMA channel normal priority register
0x685	DMA Channel Transfer Request Register, DMACREQ	DMA channel software request register
0x686	DMA Channel Status Register, DMACSTAT0	DMA channel status register0
0x687	DMA Channel Status Register, DMACSTAT1	DMA channel status register1
0x688	DMA Channel Interrupt Request Status Register, DMACIRQ	DMA channel interrupt request status register
0x689	DMA Channel Structure Base Address Register, DMACBASE	DMA channel base register
0x68A	DMA Channel Reset Register, DMACRST	DMA channel reset register
0x68B	DMA Peripheral Flow Controller Enable Register, DMAFENB	DMA peripheral flow controller enable register
0x68C	DMA Peripheral Flow Controller Disable Register, DMAFDSB	DMA peripheral flow controller disable register
See Table 44-12 See Table 44-13 See Table 44-14	DMA Channel Control Register, DMACTRL0 (DMA Channel Control Register, DMACTRLx)	Auxiliary DMA channel0 control register
See Table 44-12 See Table 44-13 See Table 44-14	DMA Channel Source Address Register, DMASARx (DMA Channel Source Address Register, DMASARx)	Auxiliary DMA channel0 source address register

Table 44-1 DMA Auxiliary Registers

Address	Name	Description
See Table 44-12 See Table 44-13 See Table 44-14	DMA Channel Destination Address Register, DMADARx (DMA Channel Destination Address Register, DMADARx)	Auxiliary DMA channel destination0 address register
N/A (memory-mapped address)	DMA Channel Linked-List Pointer Register, DMALLPx	DMA channel structure pointer register

44.2.1 XY DMA Controller

The XY DMA controller is a feature-limited version of the full DMA controller that is only available for use with the XY Memory component. The XY DMA controller supports one aux-based DMA channel and is limited to memory-to-memory transfer type operations. [Table 44-2](#) lists the registers that are accessible in the XY DMA controller.

**Note**

Only single-block transfers are supported by the XY DMA controller. Linked-list block transfers can be performed with the full DMA controller only.

Table 44-2 XY DMA Controller Auxiliary Registers

Address	Name	Description
0x680	DMA Controller Configuration Register, DMACTRL	DMA controller control register.
0x681	DMA Channel Enable Register, DMACENB	DMA channel enable register
0x682	DMA Channel Disable Register, DMACDSB	DMA channel disable register
0x686	DMA Channel Status Register, DMACSTAT0	DMA channel status register0
0x687	DMA Channel Status Register, DMACSTAT1	DMA channel status register1
0x688	DMA Channel Interrupt Request Status Register, DMACIRQ	DMA channel interrupt request status register
0x68A	DMA Channel Reset Register, DMACRST	DMA channel reset register

Table 44-2 XY DMA Controller Auxiliary Registers

Address	Name	Description
0x690	DMA Channel Control Register, DMACTRL0 (DMA Channel Control Register, DMACTRLx)	Auxiliary DMA channel0 control register
0x691	DMA Channel Source Address Register, DMASARx (DMA Channel Source Address Register, DMASARx)	Auxiliary DMA channel0 source address register
0x692	DMA Channel Destination Address Register, DMADARx (DMA Channel Destination Address Register, DMADARx)	Auxiliary DMA channel destination0 address register

44.2.2 DMA Controller Configuration Register, DMACTRL

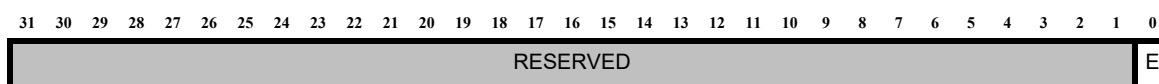
Address: 0x680

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00000000

Figure 44-1 DMACTRL Register



Use this register to enable or disable the DMA controller. When reset, the DMA controller is disabled.

When the DMA controller is disabled, any DMA request from an external peripheral or a software application is not serviced. The DMA controller should not be disabled while there are active DMA operations. Disabling the DMA controller too early may prevent outstanding DMA operations from completing.

Table 44-3 DMACTRL Field Description

Field	Bit	Description
E	0	Enable or disable the DMA controller ■ 0x1: Enable ■ 0x0: Disable

44.2.3 DMA Channel Enable Register, DMACENB

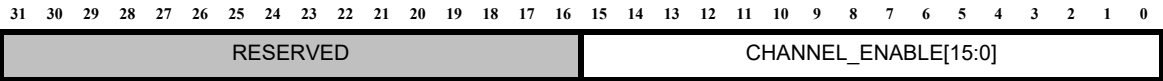
Address: 0x681

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00000000

Figure 44-2 DMACENB Register



Use this register to enable DMA channels in the DMA controller. Each bit is assigned to a channel starting with bit 0 for channel 0, bit 1 for channel 1, and so on until bit 15 for channel 15. Write 1 to a bit to enable the corresponding channel. This register can be read to determine which channel is enabled. A DMA channel can only execute a data transfer if it has been enabled. A write of 0 to this register is ignored.

Table 44-4 DMACENB Field Description

Field	Bit	Description
CHANNEL_ENABLE	[15:0]	Enable a DMA channel; each bit in this field [15:0] must be set to 1 to enable the corresponding channel

44.2.4 DMA Channel Disable Register, DMACDSB

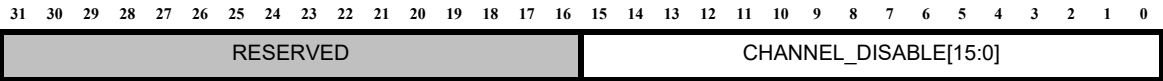
Address: 0x682

Access:

- When SecureShield 2+2 mode is not configured: W
- When SecureShield 2+2 mode is configured: W in the secure mode only

Reset: 0x00000000

Figure 44-3 DMACDSB Register



Use this register to disable DMA channels in the DMA controller. Each bit is assigned to a channel starting with bit 0 for channel 0, bit 1 for channel 1, and so on until bit 15 for channel 15. Write 1 to a bit to disable the corresponding channel. A write of 0 to this register is ignored.

Table 44-5 DMACDSB Field Description

Field	Bit	Description
CHANNEL_DISABLE	[15:0]	Disable a DMA channel; each bit in this field [15:0] must be set to 1 to disable the corresponding channel

44.2.5 DMA Channel High Priority Level Register, DMACHPRI

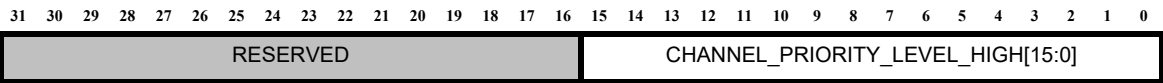
Address: 0x683

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00000000

Figure 44-4 DMACHPRI Register



Use this register to set the priority level of a DMA channel to high. By default, each channel is set to normal priority. Each bit is assigned to a channel starting with bit 0 for channel 0, bit 1 for channel 1, and so on until bit 15 for channel 15. Write 1 to a bit to set the priority level of the corresponding channel to high. The register can be read to determine which channels have high priority. A write of 0 to this register is ignored.

Table 44-6 DMACHPRI Field Description

Field	Bit	Description
CHANNEL_PRIORITY_LEVEL_HIGH	[15:0]	0x1: Set the priority level of a DMA channel to high; each bit in this field [15:0] corresponds to a DMA channel

44.2.6 DMA Channel Normal Priority Level Register, DMACNPRI

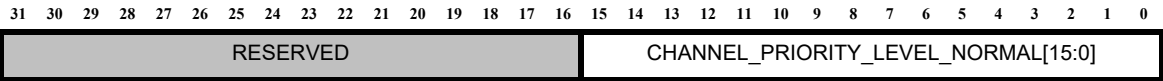
Address: 0x684

Access:

- When SecureShield 2+2 mode is not configured: W
- When SecureShield 2+2 mode is configured: W in the secure mode only

Reset: 0x00000000

Figure 44-5 DMACNPRI Register



Use this register to set the priority level of a DMA channel to normal. By default each channel is set to normal priority. Each bit is assigned to a channel starting with bit 0 for channel 0, bit 1 for channel 1, and so on until bit 15 for channel 15. Write 1 to a bit to set the priority level of the corresponding channel to normal. A write of 0 to the register is ignored.

Table 44-7 DMACNPRI Field Description

Field	Bit	Description
CHANNEL_PRIORITY_LEVEL_NORMAL	[15:0]	0x1: Change the priority of a DMA channel from high to normal; each bit in this field [15:0] corresponds to a DMA channel

Priority Selection

High priority DMA channels are given precedence over normal priority DMA channels. As a result, high priority DMA channels always complete their transfer first before any normal priority DMA channels are serviced.

DMA channels of same priority level are given equal access, as long as they are arbitrated, using the round-robin selection scheme.

44.2.7 DMA Channel Transfer Request Register, DMACREQ

Address: 0x685

Access:

- When SecureShield 2+2 mode is not configured: W
- When SecureShield 2+2 mode is configured: W in the secure mode only

Reset: 0x00000000

Figure 44-6 DMACREQ Register



Use this register to initiate a DMA transfer from software. Each bit is assigned to a channel starting with bit 0 for channel 0, bit 1 for channel 1, and so on until bit 15 for channel 15. Write 1 to a bit to initiate a data transfer on the respective DMA channel. After you have initiated a data transfer, subsequent writes to that bit of a DMA channel that is currently processing a data transfer are ignored.

Table 44-8 DMACREQ Field Description

Field	Bit	Description
TRANSFER_REQUEST	[15:0]	Initiate DMA transfer on a channel; each bit in this field [15:0] must be set to 1 to initiate a DMA data transfer on the corresponding channel.

The following steps are required to initiate a DMA transfer from a software application:

1. Enable the DMA controller using the `DMACTRL` register.
2. Enable the DMA channel on which you want to transfer data using the `DMACENB` register.
3. Define the characteristics of the block using the `DMACTRLx` register.
4. Specify the source address using the `DMASARx` register.
5. Specify the destination address using the `DMADARx` register.
6. Issue the data transfer request using the `DMACREQ` register.



Note

When setting up data transfers for a peripheral, as a best practice write the `DMACTRLx` register last in the sequence to avoid triggering a data transfer with invalid source and destination addresses.

Any DMA channel can be used for either software or peripheral controlled transfers. However, transfers initiated from software or a peripheral must be maintained under the same control until the transfer completes.

44.2.8 DMA Channel Status Register, DMACSTAT0

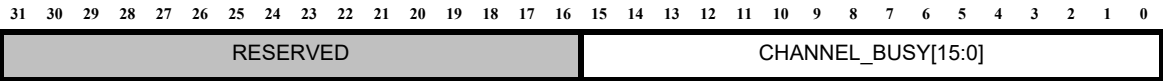
Address: 0x686

Access:

- When SecureShield 2+2 mode is not configured: R
- When SecureShield 2+2 mode is configured: R in the secure mode only

Reset: 0x00000000

Figure 44-7 DMACSTAT0 Register



Use this register to read the active status of all the DMA channels. Each bit is assigned to a channel, starting with bit 0 for channel 0, bit 1 for channel 1, and so on until bit 15 for channel 15. When a DMA channel receives a data transfer request and it is acknowledged, its respective bit in the DMACSTAT0 register is immediately set to 1 to indicate that the data transfer is in progress. When the data transfer is complete, the bit is cleared to 0.

Table 44-9 DMACSTAT0 Field Description

Field	Bit	Description
CHANNEL_BUSY	[15:0]	Read the data transfer status of each channel; each bit in this field [15:0] corresponds to a DMA channel. 0x1: channel is active; data transfer is in progress 0x0: channel is inactive

44.2.9 DMA Channel Status Register, DMACSTAT1

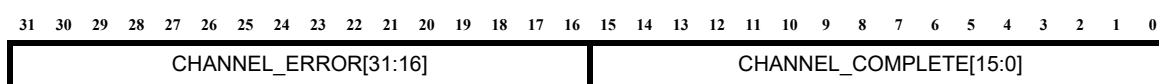
Address: 0x687

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00000000

Figure 44-8 DMACSTAT1 Register



This register indicates the completion and the error status of all DMA channels.

Bits [15:0] indicate the completion status where each bit is assigned to a channel, starting at bit 0 for channel 0, bit 1 for channel 1, and so on until bit 15 for channel 15. When a data transfer on a DMA channel has completed, the corresponding bit in the DMACSTAT1 register is set to 1. These bits are only cleared when their respective DMA channels are disabled, re-enabled, or a new data transfer is initiated.

Bits [31:16] indicate the error status where each bit is assigned to a channel, starting at bit 0 for channel 16, bit 1 for channel 17, and so on until bit 31 is for channel 31. When a memory or auxiliary access error occurs during a data transfer on a DMA channel, the corresponding bit is set to 1, and the corresponding DMA channel is immediately disabled. Error bits are cleared following an explicit write (with the target error bits to clear set to 1) to the DMACSTAT1 register. You must clear an error bit only if you have already read the register and detected that the respective error bit has been set. An error interrupt bit can also be cleared if the respective channel is reset using the DMACRST register.

Table 44-10 DMACSTAT1 Field Description

Field	Bit	Description
CHANNEL_COMPLETE	[15:0]	Read the data transfer status of each channel; each bit in this field [15:0] corresponds to a DMA channel. ■ 0x1: Data transfer is complete ■ 0x0: Channel is busy or inactive

Table 44-10 DMACSTAT1 Field Description

Field	Bit	Description
CHANNEL_ERROR	[31:16]	Indicates if an error has occurred during data transfer on a DMA channel; each bit in this field [15:0] corresponds to a DMA channel and can take the following values: ■ 0x1: Error during data transfer ■ 0x0: No error during data transfer Errors can occur due to the following reasons: ■ Auxiliary read or write to an undefined location ■ Auxiliary read from a write-only location ■ Auxiliary write to a read-only location ■ Memory read or write bus error ■ Data transfer that actively crosses a memory target boundary

44.2.10 DMA Channel Interrupt Request Status Register, DMACIRQ

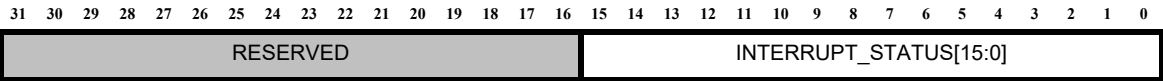
Address: 0x688

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00000000

Figure 44-9 DMACIRQ Register



 **Note** This register does not exist if `dmac_int_config==none`.

Use this register to read the interrupt request status of all the DMA channels. Each bit is assigned to a channel, starting at bit 0 for channel 0, bit 1 for channel 1, and so on until bit 15 for channel 15. When a DMA channel raises a level interrupt, the corresponding bit is set in this register. To clear an interrupt, write 1 to the channel’s corresponding bit in this register.

Table 44-11 DMACIRQ Field Description

Field	Bit	Description
INTERRUPT_STATUS	[15:0]	Read the interrupt status of each channel; each bit in this field [15:0] corresponds to a DMA channel. 0x1: the corresponding DMA channel has raised an interrupt 0x0: the corresponding DMA channel has not raised an interrupt

44.2.11 DMA Channel Structure Base Address Register, DMACBASE

Address: 0x689

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00000000

The DMACBASE register is used to define the memory location of the address pointer table which is used to locate the memory-mapped DMA descriptors. The DMA controller uses the address value in this register to locate all memory-mapped DMA channel registers. When the registers are read, the DMA transfer is immediately initiated.

 **Note** When `-sec_modes_option==true`, the DMA descriptors must be in a secure memory region.

Depending on how many DMA channels are defined, the starting address table must be aligned accordingly. This eliminates the need to allocate the maximum memory table size for a DMA controller containing 16 channels.

For example, for a two channel DMA controller, the base address of the table must be aligned to an 8-byte address boundary, for a four channel DMA controller, the base address must be aligned to a 16-byte address boundary. The following DMACBASE registers illustrates the valid portion of a register for different number of channel configurations.

Figure 44-10 DMACBASE Register for Two DMA Channels

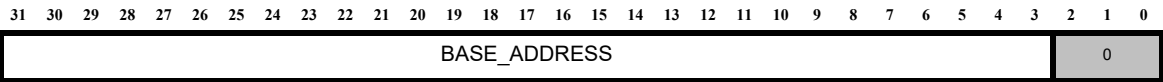


Figure 44-11 DMACBASE Register for Three or Four DMA Channels

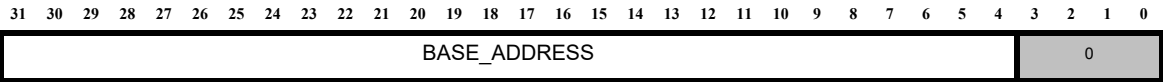


Figure 44-12 DMACBASE Register for Five to Eight DMA Channels

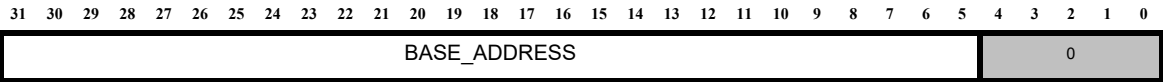


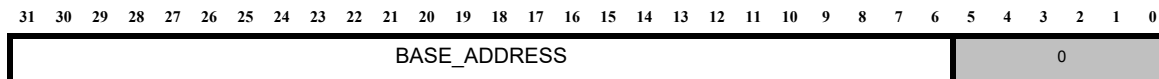
Figure 44-13 DMACBASE Register for 16 DMA Channels

Table 44-12 lists the DMA channel-specific register addresses when all the channel registers are mapped to the memory.

**Note**

The DMA channel descriptors (DSCPTRx) must be 16-byte aligned.

Table 44-12 Memory-Mapped DMA Channel-Specific Registers

Address	DMA Channel Descriptor	Description
DMACBASE+0x0	DSCPTR0	DSCPTRx defines the address location of the DMA channelx specific registers: ■ DSCPTRx: <i>DMACTRLx</i>, ■ DSCPTRx+0x4: <i>DMASARx</i>, ■ DSCPTRx+0x8: <i>DMADARx</i> ■ DSCPTRx+0xC: <i>DMALLPx</i>
DMACBASE+0x4	DSCPTR1	
DMACBASE+0x8	DSCPTR2	
DMACBASE+0xC	DSCPTR3	
DMACBASE+0x10	DSCPTR4	
DMACBASE+0x14	DSCPTR5	
DMACBASE+0x18	DSCPTR6	
DMACBASE+0x1C	DSCPTR7	
DMACBASE+0x20	DSCPTR8	
DMACBASE+0x24	DSCPTR9	
DMACBASE+0x28	DSCPTR10	
DMACBASE+0x2C	DSCPTR11	
DMACBASE+0x30	DSCPTR12	
DMACBASE+0x34	DSCPTR13	
DMACBASE+0x38	DSCPTR14	
DMACBASE+0x3C	DSCPTR15	

For DMA channels using the auxiliary registers to store channel specific registers, the associated location in the memory pointer table holds the DMALLPx register (the pointer to the next DMA descriptor in memory for linked-list operations).

Table 44-13 lists the DMA channel-specific register addresses when all the channel registers are mapped to the auxiliary register space. Note that the register addresses assigned to a channel are fixed. For example, if you have not mapped channel 0 and channel 1 to the auxiliary register space, the registers from 0x690 through 0x695 are unavailable. The linked-list point register, DMALLPx, is located in the respective offset in the memory table defined by DMACBASE.

Table 44-13 DMAC Channel Auxiliary Registers

Address	DMA Channel Data Structure
0x690	DMACTRL0
0x691	DMASAR0
0x692	DMADAR0
DMACBASE	<i>DMALLP0</i>
0x693	DMACTRL1
0x694	DMASAR1
0x695	DMADAR1
DMACBASE+0x4	<i>DMALLP1</i>
...	...
0x6BD	DMACTRL15
0x6BE	DMASAR15
0x6BF	DMADAR15

Table 44-14 lists the DMA channel-specific registers for a four-channel DMA controller with two memory-based channels and two auxiliary-based channels.

Table 44-14 DMAC Channel Registers

Address	DMA Channel Data Structure
DMACBASE	DSCPTR0
DMACBASE+0x4	DSCPTR1
DMACBASE+0x8	<i>DMALLP2 for the first auxiliary-based channel</i>
DMACBASE+0xC	<i>DMALLP3 for the second auxiliary-based channel</i>

44.2.12 DMA Channel Control Register, DMACTRLx

Address if memory-mapped:	See Table 44-12
Address if auxiliary-mapped:	See Table 44-13
Address if some DMA channels are auxiliary-mapped and some are memory-mapped:	See Table 44-14
Access:	■ When SecureShield 2+2 mode is not configured: RW ■ When SecureShield 2+2 mode is configured: RW in the secure mode only
Reset:	0x00000000

Figure 44-14 DMACTRLx Register

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
AM		I	ARB								BLOCK_SIZE								DW		DTT	RT	OP								

The DMA control register is used to define the type of DMA transfer that should be performed. A transfer, provided the channel has been enabled, can be initiated through software by writing to the DMA channel transfer request register `DMACREQ` or using an external hardware peripheral through the peripheral request input signals: `dma_req` and `dma_sreq`. The `DMACTRLx` register is always updated following the completion of the transfer.

OP

The operation field defines the type of data transfer that should be executed. Two types of data transfers are supported: Single and Linked-List.

Table 44-15 DMACTRLx OP Field Description

Field	Bit	Description
OP	[1:0]	■ 0x0: Invalid transfer ■ 0x1: Single transfer ■ 0x2: Linked-list transfer (auto-request) ■ 0x3: Linked-list transfer (manual-request)

- **Invalid transfer:** In this mode, a transfer for the respective DMA channel is considered invalid (no transfer is defined) regardless of whether the DMA channel is enabled in the `DMACENB` register. Transfer requests to an invalidate DMA channel are ignored.

- **Single transfer:** In this mode, a single data transfer is performed per request. `DMACTRLx` in the DMA descriptor is always updated during arbitration (if another channel is scheduled for processing). When the data transfer is complete, the channel control register is invalidated, this field set to 0. The `DMACTRLx` register must be written again to define and validate a new transfer. A single data transfer can be completed as the result of a single request or multiple requests.
- **Linked-list transfer (auto-request):** In this mode, the DMA channel register `DMALLPx` is used to locate the next DMA descriptor to process when the data transfer for the current DMA descriptor completes. The new DMA descriptor is automatically loaded (no data transfer request is necessary) and a data transfer is executed. Individual data transfers for each DMA descriptor are continually loaded automatically and executed until a DMA descriptor terminates the list with a single operation type. Interrupts are masked in this mode. For more information about linked lists, see [Example 44-1](#).

**Note**

Linked-list DMA descriptor base address locations must be aligned to a 16-byte address boundary.

- **Linked-list transfer (manual-request):** In this mode, the DMA channel register `DMALLPx` is used to locate the next DMA descriptor to process when the data transfer for the current DMA descriptor completes. The new DMA descriptor is loaded and a new data transfer is executed only when another data transfer request is made to the respective DMA channel. DMA descriptors are continually loaded and block transfer is executed following each transfer request until a DMA descriptor terminates the list with a single operation type. For more information about linked lists, see [Example 44-1](#).

RT

The request type field is used when a data transfer is broken down into fixed sized bursts using the arbitration field (bits [28:21]) of this register. Following the completion of each burst, the DMA controller arbitrates so that it can select and process other active DMA channels.

Table 44-16 DMACTRLx RT Field Description

Field	Bit	Description
RT	[2]	■ 0x0 : Auto request; following the initial request, the DMA channel automatically requests another fixed-sized burst when current burst completes. These requests continue until the entire block data transfer completes. - Note: Requests made by peripheral interfaces must not be set to auto mode, dma_ack and dma_done is mandatory during handshake. ■ 0x1 : Manual request; a data transfer request must be issued to the DMA channel for it to execute a fixed-sized burst. Requests must be continually issued to the DMA channel until the entire block data transfer completes. - Note: When the request type is set to manual-request and arbitration is needed, the DMACTRLx register for the active channel is only updated when another channel must be serviced or the active channel is disabled using the DMACDSB register.

DTT

This field is used to define the source and destination target types as: auxiliary or memory.

Table 44-17 DMACTRLx DTT Field Description

Field	Bit	Description
DTT	[4:3]	■ 0x0 : Memory to memory ■ 0x1 : Memory to auxiliary ■ 0x2 : Auxiliary to memory ■ 0x3 : Auxiliary to auxiliary

- **Memory to memory:** Memory to Memory transfer type is selected when performing data transfers between memories such as: system memory, ICCM0, ICCM1, DCCM, XY memory, and uncached peripheral memory. The DMA controller uses internal address decoding to access the correct memory.
- **Memory to auxiliary/auxiliary to memory:** Auxiliary data transfer types are selected when performing data transfers to and from the auxiliary space. The DMA auxiliary interface is arbitrated with the main core auxiliary interface. As a result, LR/SR instruction accesses take precedence over DMA auxiliary accesses.

Byte or half-word data transfers with auxiliary space are handled as follows: When the auxiliary space is the source, byte or half-word data must be aligned starting from the lowest significant bit of the 32-bit word. When the auxiliary space is the destination, byte and half-word data is aligned to the LSB of the 32-bit word and zero extended.

DW/INC

This field defines the data width and address increment for each DMA cell transfer.



Note

The `DMACTRLx.AM` field controls whether the source or destination address should be incremented after every DMA cell transfer.

Table 44-18 DMACTRLx DW Field Description

Field	Bit	Description
DW	[7:5]	■ 0x0: Data width is a byte and address increment is a byte ■ 0x1: Data width is a byte and address increment is 16-bits ■ 0x32 Data width is a byte and address increment is 32-bits ■ 0x3: Data width is 16-bits and address increment is 16-bits ■ 0x4: Data width is 16-bits and address increment is 32-bits ■ 0x5: Data width is 32-bits and address increment is 32-bits ■ 0x6: Clear mode, initialize a region of memory to zero. In this mode, no source memory bandwidth is consumed in the DMA data transfer. Instead, the value zero is written to the destination memory location.

BLOCK_SIZE

This field defines the size of the entire data transfer in bytes. This field is 13-bits in width allowing a maximum of 8 Kbytes of data to be transferred.

Table 44-19 DMACTRLx BLOCK_SIZE Field Description

Field	Bit	Description
BLOCK_SIZE	[20:8]	■ 0x0: one byte ■ 0x1: two bytes ■ 0x2: three bytes ■ ■ 0x1FFF: 8 Kbytes

ARB

Defines the number of DMA cells that can be transferred before the DMA controller arbitrates to determine which DMA channel should next get access to the memory bus. DMA channels can be configured to have one of two priority levels: **Normal** or **High**. High priority channels are selected by the controller over normal priority channels. Channels of the same priority are given equal access to the memory bus using a round-robin scheme. The priority setting for each channel is set in the `DMACHPRI` register. Active DMA channels request memory access to complete their DMA transfers. The controller polls all channel request lines to detect if there has been a request. If there are active high-priority channel requests, the channel with the

lowest number is selected first. If there are only normal-priority channel requests, the channel with the lowest number is selected first. The priority of channels can be changed dynamically.

**Note**

During arbitration, the DMACTRLx register in the DMA descriptor is always updated.

Table 44-20 DMACTRLx ARB Field Description

Field	Bit	Description
ARB	[28:21]	■ 0x0: No arbitration ■ 0x1: Arbitrate after 1 DMA cell transfer ■ 0x2: Arbitrate after 2 DMA cell transfers ■ 0x3: Arbitrate after 3 DMA cell transfers ■ ■ 0xFF: Arbitrate after 255 DMA cell transfers

I

The interrupt mode bit defines whether an internal or external interrupt is raised when a block transfer is complete.

Table 44-21 DMACTRLx I Bit Description

Field	Bit	Description
I	29	■ 0x0: interrupt mode is disabled; interrupts are not raised when a DMA data transfer is complete. ■ 0x1: interrupt mode is enabled; interrupts are raised when a DMA data transfer is complete.

AM

The addressing mode defines whether the source and destination addresses should be updated after a DMA cell transfer. The following addressing modes are supported:

- **Constant Addressing:** This mode uses a single source or destination address throughout the data transfer, that is, the addresses are never incremented. Typically this addressing scheme is used to connect to I/O devices that use a single address to transfer information.
- **Sequential Addressing:** This mode uses incrementing source or destination addresses to transfer a block of data in memory.

The source and destination can be individually configured to use constant or sequential addressing. [Table 44-22](#) lists the configurations available.

Table 44-22 DMACTRLx AM Field Description

Field	Bit	Description
AM	[31:30]	■ 0x0: Source and address are not incremented ■ 0x1: Source address is incremented, destination address is not incremented ■ 0x2: Source address is not incremented, destination address is incremented ■ 0x3: Source address is incremented and the destination address is incremented

Reserved fields in DMACTRLx

The DMACTRLx register includes the following reserved fields and should not be used.

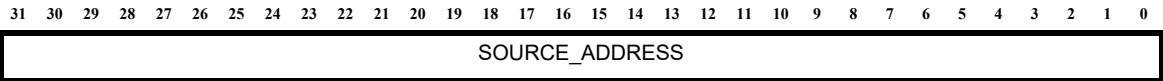
1. For XY DMA controller, the DMACTRLx [1:0] == 2 | 3 values are reserved. You cannot perform linked-list transfers with a XY DMA controller. If you do linked-list transfer with a XY DMA controller, the request is ignored and the respective channel is disabled; no error is raised.
2. Regardless of the DMA channel type, the data width or the increment field value of 7 is reserved (DMACTRLx [7:5] == 7).
3. The data transfer type field values of 1, 2, 3 are reserved (DMACTRLx [4:3] == 1 | 2 | 3) when the DMA controller is built without an auxiliary interface (used to perform data transfers to auxiliary based registers).

The DMA controller ignores transfer requests where the DMACTRLx value uses these reserved fields. The respective DMA channel is also disabled; a data transfer error is not raised in this situation.

44.2.13 DMA Channel Source Address Register, DMASARx

Address if memory-mapped:	See Table 44-12
Address if auxiliary-mapped:	See Table 44-13
Address if some DMA channels are auxiliary-mapped and some are memory-mapped:	See Table 44-14
Access:	■ When SecureShield 2+2 mode is not configured: RW ■ When SecureShield 2+2 mode is configured: RW in the secure mode only
Reset:	0x00000000

Figure 44-15 DMASARx Register



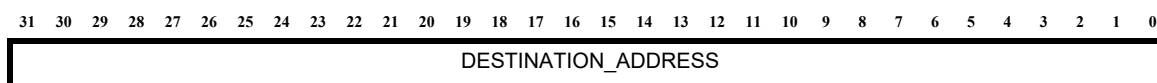
This register provides the end address location of the source memory target. The address is specified in bytes for a memory target or 32-bit data locations for an auxiliary space. The transfer block size field in the `DMACTRLx` register is subtracted from this value to calculate the true starting address location. The block size field is continually decremented until it reaches the value 0, and until the ending source address location is reached.

For example, when targeting memory, the value 5 written to this register specifies byte location 5 or a byte within the second 32-bit data word. When targeting auxiliary space, this value refers to auxiliary address location 5.

44.2.14 DMA Channel Destination Address Register, DMADARx

Address if memory-mapped:	See Table 44-12
Address if auxiliary-mapped:	See Table 44-13
Address if some DMA channels are auxiliary-mapped and some are memory-mapped:	See Table 44-14
Access:	■ When SecureShield 2+2 mode is not configured: RW ■ When SecureShield 2+2 mode is configured: RW in the secure mode only
Reset:	0x00000000

Figure 44-16 DMADARx Register



This register provides the end address location of the destination memory target. The address is specified in bytes for a memory target or 32-bit data locations for an auxiliary space. The transfer block size field in the `DMACCTRLx` register is subtracted from this value to calculate the true starting address location. The block size field is continually decremented until it reaches the value 0, and until the ending source address location is reached.

For example, when targeting memory, the value 5 written to this register specifies byte location 5 or a byte within the second 32-bit data word. When targeting auxiliary space, this value refers to auxiliary address location 5.

Calculating the source and destination transfer addresses

The end address, the last location to be accessed for a read or write, must be specified when defining a DMA data transfer.

The following pseudo code can be used to calculate the end memory address when only the burst size and required starting address are known.

```

if (byte transfer)
{
    if (address increment is a byte)
        end address = req. start address + (transfer size in bytes - 1)
    else if (address increment is a half-word)
        end address = req. start address + ((transfer size in bytes - 1) << 1)
    else if (address increment is a word)
        end address = req. start address + ((transfer size in bytes - 1) << 2)
}

```

```

if (half word transfer)
{
    if (address increment is half word)
        end address = req. start address + ((transfer size in bytes - 1) & ~0x1)
    else if (address increment is word)
        end address = req. start address + (((transfer size in bytes - 1) & ~0x1) <<1)
}

if (word transfer)
{
    end address = req. start address + ((transfer size in bytes - 1) & ~0x3))
}

```

The following pseudo code is used to calculate the end auxiliary address when only the burst size and required starting address are known.

```

If (byte transfer)
    address = start aux address + transfer size in bytes - 1

If (half word transfer)
{
    if (address increment)
        aux address = start aux address + ((transfer size in bytes - 1) >>1)
}
If (word transfer)
{
    if (address increment)
        aux address = start aux address + ((transfer size in bytes - 1) >> 2)
}

```

44.2.15 DMA Channel Linked-List Pointer Register, DMALLPx

Address if memory-mapped: See [Table 44-12](#)

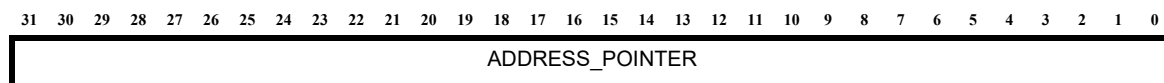
Address if auxiliary-mapped: See [Table 44-13](#)

Access:

- When SecureShield 2+2 mode is not configured: RW
- When SecureShield 2+2 mode is configured: RW in the secure mode only

Reset: 0x00000000

Figure 44-17 DMALLPx Register



Use this register when linked-list DMA block transfers are required. The address value written to this register specifies the next DMA descriptor in memory to be used after the current DMA transfer completes.

 Note

- DMA descriptors must exist in the same memory region as the DMA channel table that is defined by the DMACBASE register. Attempts to locate and point to a DMA descriptor outside of this region are regarded as undefined behavior. When `secure_modes_option==true`, the DMA descriptors must exist in the secure memory region.
- Linked-list DMA descriptor base address locations must be aligned to a 16-byte address boundary.

Example 44-1 Linked-list Transfers for Memory-Mapped DMA Channels

Table 44-23 through Table 44-26 illustrate the linked-list transfers for memory-based DMA channels.

Table 44-23 DMAC Channel Descriptors

Channel	System Memory Address Location	Descriptor Pointer Address
DMA Channel 0	DMACBASE	0xC0000 (DSCPTR0)
DMA Channel 1	DMACBASE+0x4	0x60000 (DSCPTR1)

Table 44-24 Initial State of DMA Channel Descriptor Registers for Example 44-1

Memory Address	Value
0xC0000	0x00C2FF07

Table 44-24 Initial State of DMA Channel Descriptor Registers for Example 44-1

Memory Address	Value
0xC0004	0x80000
0xC0008	0x80000
0xC000C	0xB0000

1. When the initial transfer request is made to DMA channel 0, the memory table address location defined in DMACBASE is read to get the address location of the first DMA descriptor. Address locations 0xC0000 (DMACR0), 0xC0004 (DMASAR0), and 0xC0008 (DMADAR0) are read and the defined DMA transfer executed.
2. During arbitration (where the state of DMACR0 must be saved) the location in the address table is read again to locate the address location of DMACR0. The memory location is subsequently updated with the latest information for DMACR0. The loading of the DMA descriptor and the updating of the DMACR0 register during arbitration continues until the block data transfer completes.
3. When the block transfer completes, DMACR0 is invalidated. If the operation is a linked-list operation, the DMALLP0 memory location (0xC000C) is subsequently read and then immediately written into address pointer table for DMA channel 0. If the linked-list operation is auto-request the next DMA transfer is immediately executed. If the linked-list operation is manual request, the linked-list operation is halted until another request is made to the channel.

Following the completion of the data block transfer, the linked-list operation is terminated because the DMA operation was a single transfer. The state of the memory and auxiliary registers is shown below:

Table 44-25 DMAC Channel Descriptors

Channel	System Memory Address Location	Descriptor Pointer Address
DMA Channel 0	DMACBASE	0xB0000; see Table 44-24
DMA Channel 1	DMACBASE+0x4	0x60000 (DSCPTR1)

Table 44-26 Final State of DMA Channel Descriptor Registers for Example 44-1

Memory Address	Value
0xC0000	0x00C2FF04
0xC0004	0x80000
0xC0008	0x90000
0xC000C	0xB0000
0xB0000	0x00C2FF04

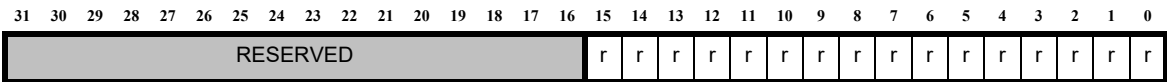
Table 44-26 Final State of DMA Channel Descriptor Registers for [Example 44-1](#)

Memory Address	Value
0xB0004	0xA0000
0xB0008	0xB0000
0xB000C	0xC0000

44.2.16 DMA Channel Reset Register, DMACRST

Address if auxiliary-mapped:	0x68A
Access:	■ When SecureShield 2+2 mode is not configured: RW ■ When SecureShield 2+2 mode is configured: RW in the secure mode only
Reset:	0x00000000

Figure 44-18 DMACRST Register



The DMACRST register is used to explicitly reset a DMA channel. Writing a 1 to the bits of this register immediately resets the respective DMA channel (including the channel’s internal state). The respective enable bits in DMACENB are also reset to 0, disabling the channels.

You can also read this register to know the reset status of a channel. The bits in this register are sticky. When a bit is set to 1, the bits remains set until the DMA channel is reset.

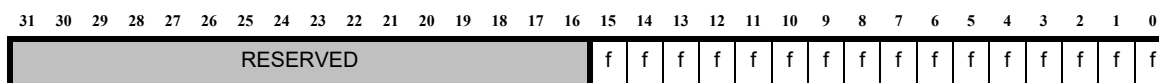
 **Note**

A software or peripheral request that is made to a DMA channel that is being reset is always ignored.

44.2.17 DMA Peripheral Flow Controller Enable Register, DMAFENB

Address if auxiliary-mapped:	0x68B
Access:	- When SecureShield 2+2 mode is not configured: RW - When SecureShield 2+2 mode is configured: RW in the secure mode only
Reset:	0x00000000

Figure 44-19 DMAFENB Register



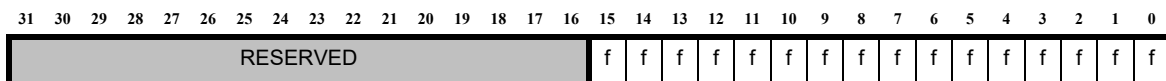
This register exists only when the full DMA controller (not the XY DMA) is used and where at least one DMA channel is connected to a DesignWare peripheral (the option `-dmac_per_if !=0`).

The DMAFENB register is used to select the DesignWare peripheral as the flow controller of a DMA channel. Each bit is assigned to one DMA channel starting with bit 0 for channel 0, bit 1 for channel 1, and so on. When you set a bit to 1, the DesignWare peripheral is the flow controller for the respective DMA channel. For example, a value of 0110 in this register indicates that the DesignWare peripheral is the flow controller for DMA channels 1 and 2.

44.2.18 DMA Peripheral Flow Controller Disable Register, DMAFDSB

Address if auxiliary-mapped:	0x68C
Access:	- When SecureShield 2+2 mode is not configured: W - When SecureShield 2+2 mode is configured: W in the secure mode only
Reset:	0x00000000

Figure 44-20 DMAFDSB Register



This register exists only when the full DMA controller (not the XY DMA) is used and where at least one DMA channel is connected to a DesignWare peripheral (the option `-dmac_per_if !=0`).

The DMAFDSB register is used to disable the peripheral as the flow controller. Each bit is assigned to one channel starting with bit 0 for channel 0, bit 1 for channel 1, and so on. When a bit is set to 1, the respective channel is configured with the ARC EM DMA as the flow controller.



Note

The bits of the DMAFENB and DMAFDSB register are only applicable to DMA channels that are connected to the DesignWare peripheral interface. The bits that correspond to the DMA channels that have the ARC EM DMA interface are unused and always return a zero when DMAFENB is read; that is, these channels only support the configuration where the DMA controller is always the flow controller.

When the DesignWare peripheral is the flow controller, the block size that is specified in the DMACTRLx register for each channel is undefined. In this configuration, the block size field should be set to the maximum possible size (8k), and the source and destination address calculated with respect to the maximum possible burst size. Only undefined bursts up to 8k are supported. To support larger undefined bursts, linked lists must be employed to process block transfers in 8k chunks.

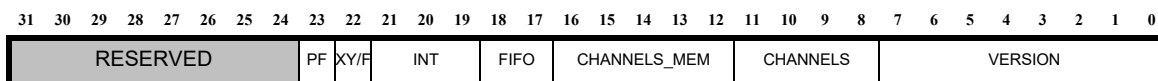
The DesignWare peripheral uses the `dma_req` and `dma_sreq` signals to control the transfer of data. When the last data block is pending transfer, the `dma_last` signal is asserted high along with the `dma_req` and/or `dma_sreq` signals. All other fields in the DMACTRLx register must be set correctly according to the type of data transfer required.

44.2.19 DMA Build Configuration Register, DMAC_BUILD

Address: 0xCD

Access: R

Figure 44-21 DMAC_BUILD Register



This register indicates the presence of the DMA controller and its configuration.

Table 44-27 DMAC_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of the DMA controller 0x1: Initial version 0x2: second version 0x3: This version includes the following updates: - New 22 indicates if the XY DMA controller or the full DMA controller is configured. - New bit 23 indicate if the DesignWare peripheral interface is configured.
CHANNELS	[11:8]	Number of DMA channels 0x0: 1 0x1: 2 0x2: 3 0xF: 16
CHANNELS_MEM	[16:12]	Defines the number of DMA channels that have their registers mapped to the auxiliary space 0x0: All DMA channels have memory-mapped registers 0x1: 1 DMA channel 0x2: 2 DMA channels 0xF: 15 DMA channels 0x10: 16 DMA channels

Table 44-27 DMAC_BUILD Field Descriptions (Continued)

Field	Bit	Description
FIFO	[18:17]	Depth of the internal FIFO 0x0: One byte 0x1: Two bytes 0x2: Reserved 0x3: Four bytes
INT	[21:19]	Interrupt configuration of the DMA controller 0x0: None 0x1: Single-Internal 0x2: Multiple-Internal 0x3: Single-External 0x4: Multiple-External
XY/F (XY/Full DMA)	[22]	Indicates the type of the DMA controller. 0x0: XY DMA controller is configured. 0x1: Full DMA controller.
PF (DMAC_PER_IF Check)	[23]	Specifies if DesignWare peripheral is configured as a flow controller. 0x0: DesignWare peripheral interface is not configured. 0x1: DesignWare peripheral interface is configured for at least one DMA channel.

Part 4

ARCV2 Scalar DSP Extensions

In this part:

- [Scalar DSP Introduction](#)
- [Data Organization and Addressing](#)
- [DSP Auxiliary Interface](#)
- [Scalar DSP Instruction Set Details](#)

45

Scalar DSP Introduction

This document is intended for programmers of the ARCV2 ISA (Instruction-Set Architecture) and the ARCV2DSP ISA. The ARCV2 ISA comprises a mandatory set of baseline features, together with a collection of optional extensions to the ISA. This document covers all aspects of the ARCV2 ISA.

The ARCV2DSP is an extension of the ARCV2 ISA. The ARCV2DSP includes over 100 new DSP instructions for accelerating signal processing algorithms, including vector and complex MUL/MAC operations. The ARCV2DSP includes:

- a unified, single-cycle 32×32 MUL/MAC unit with 40-bit/72-bit accumulators.
- fractional support (Q31 and Q15 data types) with saturating arithmetic, rounding and non-rounding instructions, as well as divide and square root instructions to support filtering, fast Fourier transform (FFT), and other signal processing algorithms.
- vector operations to process multiple data values in a single operation.

45.1 Key Features of the ARCV2 ISA DSP Extensions

- Basic Saturating Arithmetic instructions
- Vector Unpacking instructions
- Vector ALU instructions
- Accumulator instructions
- Vector SIMD 16×16 MAC instructions
- Dual Inner Product 16×16 MAC instructions
- 32×32 and 16×16 MAC instructions
- Dual 16×8 MAC instructions
- Single-cycle Complex multiply, conjugated multiply, and FFT instructions
- 32×16 MAC instructions
- Accumulator shifting allowed in the $[-72:72]$ range.
- A small address generation unit (AGU)

45.2 Programmer's Model

The programmer's model is common to all implementations of the ARCV2-based processors and allows upward compatibility of code for software that has been compiled for similarly-configured ARCV2 processors.

45.3 Configurability

The ARCV2 architecture offers several well-defined methods for configuring the programming model and instruction set. In addition, each processor within the ARCV2 family offers well-defined methods for configuring additional core component, and to select from the available micro-architectural implementation options to choose an appropriate trade-off between performance, complexity, operating frequency, and energy consumption.

Table 45-1 DSP Configuration Options

Option Name	ARCHitect Option	Option Values	Description
DSP			
Implementation	-dsp_impl	inferred optimized	Infer the datapath components from Verilog or optimized using carry-save components for timing.
ITU Support	-dsp_itu	false true	Include support for pre-accumulate shifting required by ITU codecs ■ true: supported ■ false: insupported
Divide and Square Root	-dsp_divsqrt	none radix4	Enable support for divide and square-root instructions. radix4 computes two result bits per clock cycle
Wide DSP support	-dsp_wide	false true	If true, enable DSP instructions with 64 source operands.

46

Data Organization and Addressing

This chapter describes the data organization and addressing used by the ARCV2-based DSP extensions.

46.1 DSP Data Formats

The set of DSP extension instructions, enabled by the HAS_DSP option, use the following data formats.

46.1.1 8-Bit Signed Integer Vector

A 32-bit data format contains four 8-bit signed integers. Each signed integer (8 bits) includes one sign bit (most significant bit) and 7 bits (6:0) representing the 2's complement of the integer value. The most significant bit of an 8-bit signed integer is the sign bit.

Table 46-1 8-Bit Signed Integer Vector

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
s	2's complement							s	2's complement							s	2's complement							s	2's complement						

46.1.2 16-Bit Signed Integer

In a 32-bit data format, the 16-bit signed integer is represented by the least significant 16 bits. Of the least significant 16 bits, bit [15] is the sign bit and bits [14:0] represent the 2's complement of the integer value. The most significant 16 bits of the 32 bits are ignored.

The 16-bit signed integer is always aligned to an even memory address.

Table 46-2 16-Bit Signed Integer

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
Ignored																s	2's complement														

46.1.3 16-Bit Unsigned Integer

In a 32-bit data format, the 16-bit unsigned integer is represented by the least significant 16 bits. The most significant 16 bits of the 32 bits are ignored.

The 16-bit unsigned integer is always aligned to an even memory address.

Table 46-3 16-Bit Unsigned Integer

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
Ignored																Unsigned Integer															

46.1.4 16-Bit Signed Fractional

Scalar 16-bit signed fractional data can be stored in the most significant part of a register [31:16] or the least significant part of a register [15:0]. Therefore, a 32-bit operand can contain a vector of two 16-bit signed fractions, as described in Section 46.1.7. Operations involving scalar 16-bit fractional source operands come in two versions:

- Instructions whose mnemonic has suffix 'L', take a Q15 scalar operand from the least significant 16 bits of a 32-bit operand, for example, MPYWHFL.
- Instructions whose mnemonic has suffix 'M', take a Q15 scalar operand from the most significant 16 bits of a 32-bit operand, for example, MSUBWHFM.

The maximum representable value for a 16-bit signed fraction is 0.999969 and minimum value is -1.0.

46.1.5 16-Bit Signed Integer Vector

In a 32-bit data format, the 16-bit signed integer vector packs two 16-bit signed integers. Each signed integer (16 bits) includes one sign bit (most significant bit) and 15 bits representing the 2's complement of the integer value. The most significant bit of a 16-bit signed integer is the sign bit.

The 16-bit signed integer is always aligned to an even memory address.

Table 46-4 16-Bit Signed Integer Vector

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
s	2's complement														s	2's complement															

46.1.6 16-Bit Unsigned Integer Vector

In a 32-bit data format, the 16-bit unsigned integer vector packs two 16-bit unsigned integers. Each unsigned integer represent the 16-bit 2's complement of the integer.

The 16-bit unsigned integer vector is always aligned to an even memory address.

Table 46-5 16-Bit Unsigned Integer Vector

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
2's complement																2's complement															

46.1.7 16-Bit Signed Fractional Vector

A 32-bit data format contains two 16-bit signed fractional vectors. In each 16-bit vector, the most significant bit represents the sign bit and the least significant 15 bits represent the 2's complement of the fractional value.

The maximum representable value is 0.999969 and minimum value is -1.

The 16-bit signed fractional data is always aligned to an even memory address.

Table 46-6 16-Bit Signed Fractional Vector

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
S	2's Complement Fractional Value															S	2's Complement Fractional Value														

46.1.8 Complex 16+16 Bit Signed Fractional Data

A 32-bit data format contains two 16-bit signed fractional vectors—a 16-bit signed imaginary component and a 16-bit signed real component. In each 16-bit vector, the most significant bit represents the sign bit and the least significant 15 bits represent the 2's complement of the fractional value.

The maximum representable value is 0.999969 and minimum value is -1.

The complex data is always aligned to an even memory address.

Table 46-7 Complex 16+16 Bit Complex Signed Fractional Data

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
S	2's Complement Imaginary Fractional Value															S	2's Complement Real Fractional Value														

46.1.9 32-Bit Signed Integer

In a 32-bit signed integer, the most significant bit (bit 31) is the sign bit and the least significant bits (30:0) represent the 2's complement of the integer value.

The 32-bit signed integer is always aligned to a quad byte memory address.

Table 46-8 32-Bit Signed Integer

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
S	2's complement																														

46.1.10 32-Bit Unsigned Integer

In a 32-bit unsigned integer, bits (31:0) represent the 2's complement of the integer value.

The 32-bit unsigned integer is always aligned to a quad byte memory address.

Table 46-9 32-Bit Unsigned Integer

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
Unsigned Integer																															

46.1.11 32-Bit Signed Fractional

In a 32-bit signed fraction, the most significant bit, (bit 31), is the sign bit and the least significant bits (30:0) represent the 2's complement of the fractional value.

The 32-bit signed fraction is always aligned to a quad byte memory address.

Table 46-10 32-Bit Signed Fractional Data

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
S		2's Complement Fractional Value																													

46.1.12 Accumulators

The DSP component uses the following auxiliary registers for DSP operations:

- “ACC0_LO” on page 1292
- “ACC0_HI” on page 1294
- “ACC0_GLO” on page 1293
- “ACC0_GHI” on page 1295

The ACC0_LO register is a mirror of the r58 register in little-endian mode and the r59 register in big-endian mode. The auxiliary register ACC0_GLO is used to store the guard bits when the ACC0_LO register overflows.

Similarly, the ACC0_HI register is a mirror of the r59 register in little-endian mode and the r58 register in big-endian mode. The auxiliary register ACC0_GHI is used to store the guard bits when the ACC0_HI register overflows.

46.1.12.1 Vector Accumulators

The ACC0_LO, ACC0_HI, ACC0_GLO and ACC0_LO, and ACC0_GHI and ACC0_HI are the vector accumulators. The format of these accumulators is dependent on the following configuration options:

- Whether the guard bits are enabled (DSP_CTRL==1)
- Whether pre-accumulation is enabled (DSP_CTRL.PA==1)

The vector accumulators are used in the following operations:

Operation	ACC0_GHI GUARD	ACC0_HI	ACC0_GLO GUARD	ACC0_LO
2x32 bits		val1[31:0]		val0[31:0]
1x64 bits		val[63:32]		val[31:0]
1x32b		val[31:0]		
1x40b	val[39:32]	val[31:0]		
2x40b	val1[39:32]	val1[31:0]	val0[39:32]	val0[31:0]

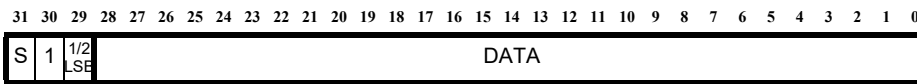


Note

The 1x72 bit operation is explained in section “Wide Accumulator” on page 1287.

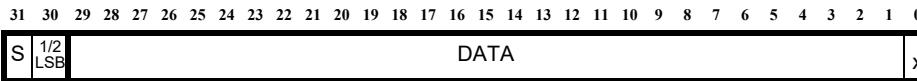
When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==0`, [Figure 46-1](#) represents the vector accumulators (`ACC0_LO` and `ACC0_HI`).

Figure 46-1 Vector Accumulator When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==0`



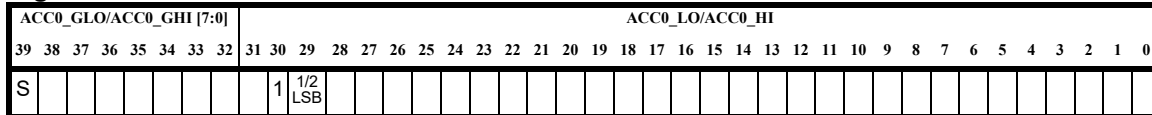
When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`, [Figure 46-2](#) represents the vector accumulators (`ACC0_LO` and `ACC0_HI`). In this mode, the LSB is set only on saturation.

Figure 46-2 Vector Accumulator When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`



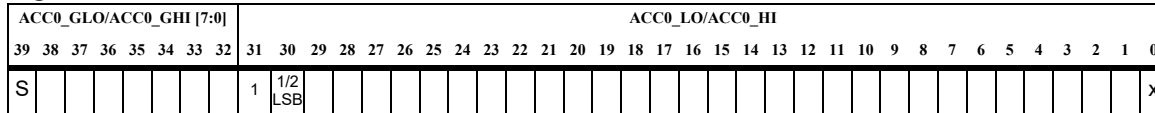
When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==0`, [Figure 46-3](#) represents the vector accumulators.

Figure 46-3 Vector Accumulator When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==0`



When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`, [Figure 46-4](#) represents the vector accumulators. In this mode, the LSB is set only on saturation.

Figure 46-4 Vector Accumulator When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`



46.1.12.2 Wide Accumulator

The wide accumulator comprises of the `ACC0_GHI`, `ACC0_HI`, and `ACC0_LO` registers. The wide accumulator is used in the following accumulation operations:

Operation	ACC0_GHI GUARD	ACC0_HI	ACC0_LO
1x72 bits	val[71:64]	val[63:32]	val[31:0]

The format of these accumulators is dependent on the following configuration options:

- Whether the guard bits are enabled (`DSP_CTRL==1`)
- Whether pre-accumulation is enabled (`DSP_CTRL.PA==1`)

When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==0`, [Figure 46-5](#) represents the wide accumulator.

Figure 46-5 Wide Accumulator When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==0`

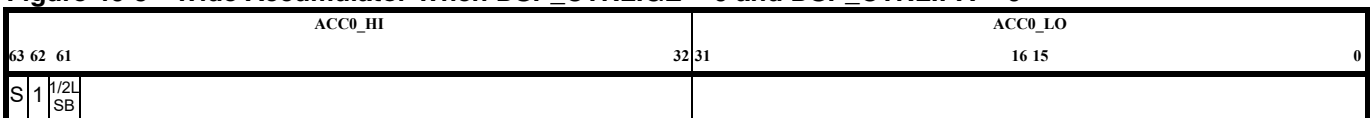


Figure 46-6 Wide Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==1

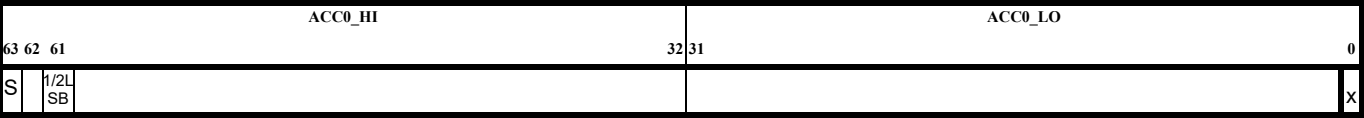


Figure 46-7 Wide Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==0

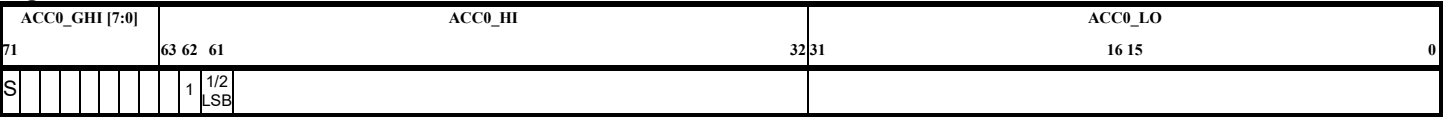
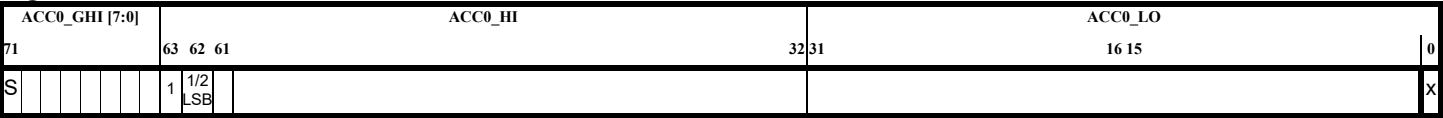


Figure 46-8 Wide Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==1



If guard bits are enabled, `DSP_CTRL.GE==1`, the accumulators saturate only if the guard bits overflow. If guard bits are disabled, `DSP_CTRL.GE==0`, the accumulators saturate as soon as the accumulator overflows.

Table 46-11 provides the list of instructions that you can use to shift, set, read, divide, and compute the square root of the accumulators:

	Type	Major	Minor	F bit	Description
ASLACC	ZOP	0x05	0x00	0	Shift the accumulator left arithmetically
ASLSACC	ZOP	0x05	0x01	0	Shift the accumulator left arithmetically; the result is saturated
FLS	ZOP	0x05	0x04	1	Copy accumulator status flags to the STATUS32 register
J	SOP	0x05	0x18	0	Read the accumulator
DIVU	ZOP	0x05	0x03	0	Divide the accumulator
SR	ZOP	0x05	0x02	0	Compute the square root of the accumulator

Table 46-11 Accumulator Instructions

	Type	Major	Minor	F bit	Description
NORMH, NORMW	SOP	0x05	0x19	0	Normalize the accumulator
SETcc	DOP	0x05	0x0d	1	Set the accumulator

47

DSP Auxiliary Interface

47.1 DSP Auxiliary Registers

The Digital Signal Processing (DSP) component includes a set of hardware and instructions that can be used to minimize the processor loading when executing entry-level DSP applications, such as basic audio processing.

The DSP unit contains the following auxiliary registers:

Table 47-1 DSP Auxiliary Registers

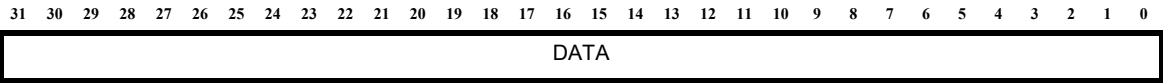
Address	Auxiliary Register Name	Description
0x580	Accumulator Low Register, ACC0_LO	Low accumulator register.
0x581	Low Accumulator Guard and Status Register, ACC0_GLO	Low accumulator guard and status register.
0x582	Accumulator High Register, ACC0_HI	High accumulator register.
0x583	High Accumulator Guard and Status Register, ACC0_GHI	High accumulator guard and status register.
0x598	DSP Butterfly Instructions Data Register, DSP_BFLY0	Register that stores the implicit complex A0 operand that is used in the butterfly instructions.
0x59E	DSP FFT Control Register, DSP_FFT_CTRL	Register that controls the scaling behavior of the complex FFT butterfly instructions.
0x59F	DSP Control Register, DSP_CTRL	Register to control the rounding and guard bit behavior.

47.1.1 Accumulator Low Register, ACC0_LO

Address: 0x580

Access: rw

Figure 47-1 ACC0_LO Register



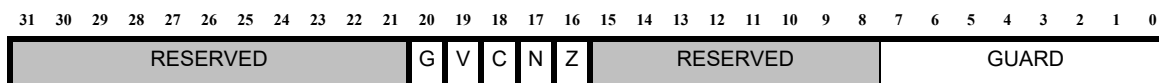
This register holds the lower 32 bits of the result from a vector accumulate operation. For complex math operations, this register holds the real part of a complex operand in a little-endian mode and the imaginary part of the complex operand in a big-endian mode. For operations that involve a wide accumulator, this register holds the least significant 32 bits of the wide accumulator.

47.1.2 Low Accumulator Guard and Status Register, ACC0_GLO

Address: 0x581

Access: rw

Figure 47-2 ACC0_GLO Register



This register holds the guard bits and the status flags of the ACC0_LO accumulator. This register is not used in the wide accumulator operations.

Table 47-2 ACC0_GLO Field Descriptions

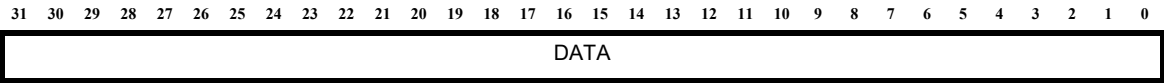
Field	Bit	Description
GUARD	[7:0]	Guard bits; overflow bits of the low vector accumulator. For example, if a the low vector accumulator (ACC0_LO) overflows, this field stores the overflow bits (up to 8 bits). This field is valid only if DSP_CTRL.GE==1.
Z	[16]	Zero flag; this bit is set only by the fractional MAC instructions when the accumulator (ACC0_LO) is zero.
N	[17]	Negative status flag; set only by the fractional MAC instructions when the accumulator (ACC0_LO) is negative.
C	[18]	Carry status flag; this carry flag is always cleared (0).
V	[19]	Overflow status flag; set only by the fractional MAC instructions if the accumulator (ACC0_LO) has overflowed on the most recent instruction.
G	[20]	Indicates if the guard bits are actively used in the fractional instructions. A fractional MAC operation sets this bit only if the GUARD field bits are not equal to the sign bit of the accumulator (ACC0_LO).

47.1.3 Accumulator High Register, ACC0_HI

Address: 0x582

Access: rw

Figure 47-3 ACC0_HI Register



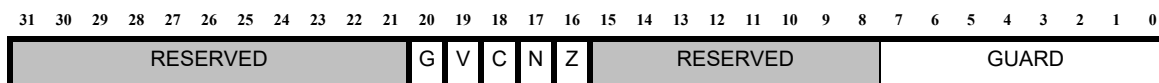
This register holds the higher 32 bits of the result from a vector accumulate operation. For complex math operations, this register holds the imaginary part of a complex operand in a little-endian mode and the real part of the complex operand in a big-endian mode. For operations that involve a wide accumulator, this register holds the most significant 32 bits of the wide accumulator.

47.1.4 High Accumulator Guard and Status Register, ACC0_GHI

Address: 0x583

Access: rw

Figure 47-4 ACC0_GHI Register



This register holds the guard bits and the status flags for the ACC0_HI accumulator.

Table 47-3 ACC0_GHI Field Descriptions

Field	Bit	Description
GUARD	[7:0]	Guard bits; overflow bits of the high vector accumulator. For example, if a the low vector accumulator (ACC0_HI) overflows, this field stores the overflow bits (up to 8 bits). This field is valid only if DSP_CTRL.GE==1.
Z	[16]	Zero flag; this bit is set only by the fractional MAC instructions when the accumulator (ACC0_HI) is zero.
N	[17]	Negative status flag; set only by the fractional MAC instructions when the accumulator (ACC0_HI) is negative.
C	[18]	Carry status flag; this carry flag is always cleared (0).
V	[19]	Overflow status flag; set only by the fractional MAC instructions if the accumulator (ACC0_HI) has overflown on the most recent instruction.
G	[20]	Indicates if the guard bits are actively used in the fractional instructions. A fractional MAC operation sets this bit only if the GUARD field bits are not equal to the sign bit of the accumulator (ACC0_HI).

47.1.5 DSP Butterfly Instructions Data Register, DSP_BFLY0

Address: 0x598

Access: rw

Figure 47-5 DSP_BFLY0 Register in Little-Endian Mode

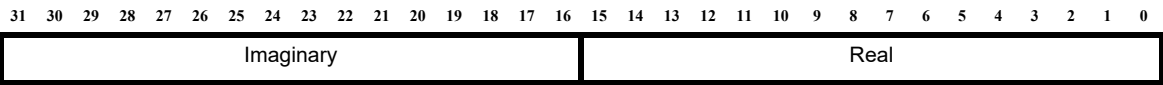
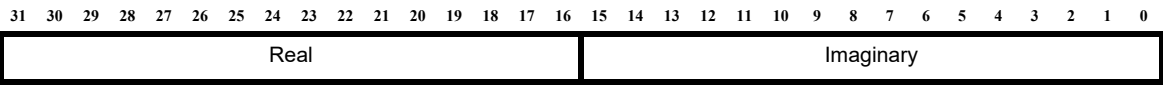


Figure 47-6 DSP_BFLY0 Register in Big-Endian Mode



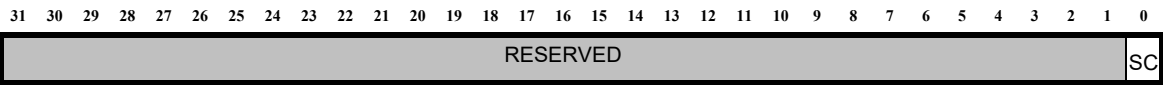
This register stores the implicit complex A0 operand that is used in the butterfly instructions. The DSP_BFLY0 register consists of the real and imaginary components of a 16+16 bit complex fractional number. The CBFLYH1R instruction can modify this register. The CBFLYH0R and CBFLYH1R instructions can implicitly read this register. This register is present in the processor build only if dsp_complex==true.

47.1.6 DSP FFT Control Register, DSP_FFT_CTRL

Address: 0x59E

Access: rw

Figure 47-7 DSP_FFT_CTRL Register



This register controls the scaling behavior of the complex FFT butterfly instructions.

This register is present in the processor build only if `dsp_complex==true`.

Table 47-4 DSP_FFT_CTRL Field Descriptions

Field	Bit	Description
SC	[0]	Controls the scaling of the complex results 0x0: no scaling 0x1: the butterfly instructions (CBFLYHF0R and CBFLYH1R) scale the complex result by right-shifting the result by one bit.

47.1.7 DSP Control Register, DSP_CTRL

Address: 0x59F

Access: rw

Figure 47-8 DSP_CTRL Register



This register controls the rounding and guard bit behavior of the DSP multiply and accumulate operations.

Table 47-5 DSP_CTRL Field Descriptions

Field	Bit	Description
RM	[1:0]	Indicates the rounding mode If <code>dsp_accshift==full</code> 0x0/1: round down to the nearest representable number 0x2: round up; add 1/2 LSB to the result and truncate 0x3: convergent rounding; rounding to the nearest even. Convergent rounding is the same as rounding up except in the following case: When the 1/2 LSB is set and all lower significant bits are cleared, convergent rounding rounds up only if the LSB is set. If <code>dsp_accshift==limited</code> - RM[0] bit always returns zero. 0: no rounding; round down to the nearest representable number 2: round up; add 1/2 LSB to the result and truncate
GE	[2]	Enable guard bits 0x0: disable guard bits; the wide accumulator saturates at 64 bits and the vector accumulators saturate at 32 bits 0x1: enable guard bits; the wide accumulator saturates at 72 bits and the vector accumulators saturate at 40 bits

Table 47-5 DSP_CTRL Field Descriptions (Continued)

Field	Bit	Description
PA	[3]	Enable pre-accumulation shift mode If <code>dsp_itu==true</code> 0x0: post accumulation. The fractional product is not shifted before accumulation. The accumulator is of type 1Q30. 0x1: pre accumulation. The fractional product is shifted by one position to the left before accumulation. The accumulator value in this case is of type Q31 with <code>LSB==0</code>. If <code>dsp_itu==false</code> - The PA bit is fixed to 0; pre-accumulation shift mode is not supported.
SE	[4]	Enable saturation for fractional instructions 0x0: disable saturation for guarded operations with fractional instructions 0x1: enable saturation for guarded operations with fractional instructions
SAT	[16]	Sticky saturation status 1: indicates that a DSP operation is saturated in one of the following cases: - Accumulator saturation - Result saturation - Multiply pre-accumulate saturation when <code>DSP_CTRL.GE==0</code> and <code>DSP_CTRL.PA==1</code> This bit is a sticky flag. You must explicitly write a 0 to this bit to clear this flag.

47.2 Build Configuration Registers

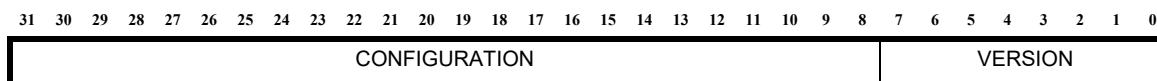
The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

47.2.1 DSP Build Configuration Register, DSP_BUILD

Address: 0x7A

Access: R

Figure 47-9 DSP_BUILD Register



This configuration register indicates the version and configuration of the DSP component. Applications and software can read this register to detect the DSP configuration.

Table 47-6 DSP_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	DSP version number 0x20: initial version 0x21: current version with XY support
CONFIGURATION	[9:8]	Indicates support for the divide and square-root instructions 0x0: DIV/REM instructions are not supported. 0x1: Radix-2 bit-serial DIV, DIVU, REM and REMU implementation. 0x2: Radix-4 fast DIV, DIVU, REM and REMU implementation.
CONFIGURATION	[10]	Indicates support for the single-cycle complex instructions 0x0: Support for two-cycle complex MAC hardware; this option does not include the complex butterfly hardware. 0x1: Support for single-cycle complex MAC, conjugated multiply, and FFT butterfly hardware.
CONFIGURATION	[12:11]	Indicates support for the accumulator shift 0x1— Shift accumulator in the range [-8:8]. 0x2— Shift accumulator in the range [-72:72].
CONFIGURATION	[13]	Indicates support for the ITU pre-accumulate support. 0x0: Preaccumulate shifting hardware is not supported. 0x1: Preaccumulate shifting hardware is supported
CONFIGURATION	[31:14]	Reserved. Read as 0.

48

Scalar DSP Instruction Set Details

48.1 Notations Used in the DSP Instructions

- b0 (lower), b1 (first), b2 (second), and b3 (upper) represent the 8-bit elements of a vector.
- h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit vector.
- w0 (lower) and w1 (upper) represent the 32-bits elements of a 64-bit vector.
- A, B, C are used to denote 64-bit operands (use a register pair). A register pair always starts with an even register. For example, for a register pair operation, r0 indicates that both r0 and r1 are used for the operation.
- For complex DSP instructions, r is used to indicate the real part of the operand, and i is used to indicate the imaginary part of the operand.

48.2 List of Instructions

Table 48-1 summarizes the 32-bit alongside the 16-bit instructions supported by the ARCV2 ISA.

Table 48-1 List of Instructions

32-Bit Instructions	
Instruction	Operation
ABSS	Absolute saturated 32-bit value
ABSSH	Absolute saturated 16-bit value
ADCS	Saturating 32-bit add with carry in
ADDS	Saturating 32-bit addition
ASLACC	Arithmetic shift left accumulator
ASLS	Saturating arithmetic shift left
ASLSACC	Saturating arithmetic shift left accumulator
ASRS	Shift right with saturating if value is negative

Table 48-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
ASRSR	Shift right rounding and saturating if value is negative
CBFLYHF0R	16 bit + 16 bit FFT butterfly, first half; return first result
CBFLYHF1R	16 bit + 16 bit FFT butterfly, second half; return first result with rounding
CMACCHFR	16 bit + 16 bit complex signed fractional conjugated multiplication and accumulation with rounding
CMACCHNFR	16 bit + 16 bit complex signed fractional conjugated multiplication and accumulation with rounding, and returned without a fractional shift
CMACHFR	16 + 16 bit complex signed fractional multiplication and accumulation with rounding
CMACHNFR	16 bit + 16 bit complex signed fractional multiplication and accumulation with rounding
CMPYCHFR	16 bit + 16 bit complex signed fractional conjugated multiplication with rounding
CMPYCHNFR	16 bit + 16 bit complex signed fractional conjugated multiplication with rounding, and returned without a fractional shift
CMPYHFMR	16 + 16 bit complex signed fractional multiplication with rounding
CMPYHNFR	16 bit + 16 bit complex signed fractional multiplication with rounding
DIV	2's complement integer division
DIVF	Signed 32-bit fractional division
DIVU	Unsigned integer Divide
DMACH	Dual 16x16 multiply and accumulate
DMACHBL	Dual vector 16 bitx8 bit signed integer multiplication and accumulation; lower bytes
DMACHBM	Dual vector 16 bitx8 bit signed integer multiplication and accumulation; higher bytes
DMACHF	Dual vector 16 bit signed fractional multiplication and accumulation
DMACHFR	Dual vector 16 bit signed fractional rounding multiplication and accumulation
DMACHU	Dual unsigned 16x16 multiply and accumulate
DMPYH	Dual 16x16 multiplication
DMPYHBL	Dual vector 16 bitx8 bit signed integer multiplication; lower bytes
DMPYHBM	Dual vector 16 bitx8 bit signed integer multiplication; higher bytes
DMPYHF	Dual vector 16 bit signed fractional multiplication
DMPYHFR	Dual vector 16 bit signed fractional rounding multiplication

Table 48-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
DMPYHU	Dual unsigned 16x16 multiplication
DMPYHWF	Dual vector 16 bit signed fractional multiplication and 32-bit write-back
FLAGACC	Copy accumulator status flags to STATUS32
GETACC	Read accumulator
MAC	32x32 multiply and accumulate
MACD	32x32 multiply and accumulate, double result
MACDF	32 bit signed fractional saturating multiplication and accumulation; 64-bit result
MACDU	Unsigned 32x32 multiply and accumulate, double result
MACF	32 bit signed fractional saturating multiplication and accumulation
MACFR	32 bit signed fractional saturating an rounding multiplication and accumulation
MACU	Unsigned 32x32 multiply and accumulate
MACWHFL	Signed 32x16 fractional multiplication and accumulation
MACWHFLR	Signed 32x16 fractional multiplication and accumulation with rounding
MACWHFM	32x16 bit signed fractional multiplication and accumulation, higher signed 16 bits
MACWHFMR	32x16 bit signed fractional multiplication and accumulation with rounding, higher signed 16 bits
MACWHL	32x16 bit signed multiplication and accumulation, lower signed 16 bits
MACWHKL	Signed 32x16 integer multiplication and accumulation and shift-right the lower bits
MACWHKUL	Unsigned 32x16 integer multiplication and accumulation and shift-right the lower bits
MACWHUL	32x16 bit unsigned multiplication and accumulation, lower signed 16 bits
MODIF	Update an address pointer based on a modifier value
MODAPP	Update the pointer without accessing the memory.
MPY	32 x 32 Signed Multiply (lsw)
MPY_S	Signed 32 x 32 multiplication. Return the least-significant 32 bits of the result.
MPYM MPYH	32 x 32 Signed Multiply (msw)
MPYD	32x32 multiplication, double result
MPYDF	32 bit signed fractional saturating multiplication; 64-bit result
MPYDU	Unsigned 32x32 multiplication, double result

Table 48-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
MPYF	32 bit signed fractional saturating multiplication
MPYFR	32 bit signed fractional saturating an rounding multiplication
MPYMU MPYHU	32 x 32 Unsigned Multiply (msw)
MPYU	32 x 32 Unsigned Multiply (lsw)
MPYUW	16 bit unsigned integer multiplication
MPYUW_S	16 bit unsigned integer multiplication
MPYW	16 X 16 signed multiply
MPYW_S	16 X 16 signed multiply
MPYWHFL	32x16 bit signed fractional multiplication, lower signed 16 bits
MPYWHFLR	32x16 bit signed fractional multiplication with rounding, lower signed 16 bits
MPYWHFM	32x16 bit signed fractional multiplication, higher signed 16 bits
MPYWHFMR	32x16 bit signed fractional multiplication with rounding, higher signed 16 bits
MPYWHL	32x16 bit signed multiplication, lower signed 16 bits
MPYWHKL	Signed 32x16 integer multiplication, and shift-right the lower bits
MPYWHKUL	Unsigned 32x16 integer multiplication, and shift-right the lower bits
MPYWHUL	32x16 bit unsigned multiplication, lower signed 16 bits
MSUBDF	32 bit signed fractional saturating subtraction; 64-bit result
MSUBFR	32 bit fractional saturating and rounding subtraction
MSUBF	32 bit fractional saturating subtraction
MSUBWHFL	Signed 32x16 fractional multiplication and subtraction
MSUBWHFLR	Signed 32x16 fractional multiplication and subtraction with rounded result
MSUBWHFM	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand
MSUBWHFMR	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand with rounded result
NEGS	Negative saturated 16-bit value
NEGSH	Negative saturated 32-bit value

Table 48-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
NORMACC	Normalize the accumulator
REM	Signed Integer Remainder
REMU	Unsigned Integer Remainder
RNDH	Rounding and saturating a 32-bit value to a 16-bit value
SATH	Saturating 32-bit value to a 16-bit value
SATF	Saturate a register based on the STATUS32.N and STATUS32.V flags
SBCS	Signed subtraction with saturation and carry in
SBCS	Subtract with carry
SETACC	Set the accumulator
SQRT	Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result
SQRTF	Compute the square-root of an unsigned Q31 fraction, yielding a Q31 fractional result
SUBS	Saturating 32-bit subtraction
VABS2H	Two way 16 bit vector absolute
VABSS2H	Two way 16 bit vector saturating absolute
VADD2H	Dual 16-bit vector addition
VADD4B	Four way add
VADDS2H	Two way vector saturating add 16 bits
VADDSUB2H	Dual 16-bit vector add and subtract
VADDSUBS2H	Two way vector saturating add subtract
VALGN2H	Two-way 16-bit vector align
VASL2H	Two way arithmetic shift left 16 bits
VASLS2H	Two way saturating arithmetic shift left 16 bits
VASR2H	Two way arithmetic shift right 16 bits
VASRS2H	Two way arithmetic saturating shift right 16 bits
VASRSR2H	Two way arithmetic saturating and rounding shift right 16 bits
VEXT2BHL	Vector extend the lower two bytes to the higher two bytes
VEXT2BHL	Compose a dual Q15 vector by expanding the lower two Q7 bytes of an operand

Table 48-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
VEXT2BHM	Vector extend the higher two bytes to the lower two bytes
VEXT2BHMf	Compose a dual Q15 vector by expanding the higher two Q7 bytes of an operand
VLSR2H	Two way vector logic shift right 16 bits
VMAC2H	Dual 16x16 vector multiplication and accumulation
VMAC2HF	Two way vector 16 bit saturating fractional multiplication and accumulation
VMAC2HFR	Two way vector 16 bit saturating and rounding fractional multiplication and accumulation
VMAC2HNFR	Two way vector 16 bit saturating and rounding fractional multiplication and accumulation
VMAC2HU	Dual unsigned 16x16 vector multiplication and accumulation
VMAX2H	Two way vector maximum 16 bits
VMIN2H	Two way vector minimum 16 bits
VMPY2H	Dual 16x16 vector multiplication
VMPY2HF	Two way vector 16 bit saturating fractional multiplication
VMPY2HFR	Two way vector 16 bit saturating and rounding fractional multiplication
VMPY2HU	Dual unsigned 16x16 vector multiplication
VMPY2HWF	Two way vector 16 bit saturating fractional multiplication; two way 32 bit write-back
VMSUB2HF	Two-way vector 16x16 signed fractional multiply-subtract, saturating
VMSUB2HFR	Two-way vector 16x16 signed fractional multiply-subtract, saturating and rounding
VMSUB2HNFR	Two-way vector 16x16 signed fractional multiply-subtract, saturating and rounding, and returned without a fractional shift
VNEG2H	Two way 16 bit vector negative
VNEGS2H	Two way 16 bit vector saturating negative
VNORM2H	Two way 16 bit vector normalization
VPACK2HL	Compose the destination operand from the lower 16-bits of the source operands
VPACK2HM	Compose the destination operand from the higher 16-bits of the source operands
VPACK2HBL	Pack the lower two bytes of a vector from the lower 8-bits of two 16-bit vectors of an operand
VPACK2HBLF	Pack the lower two bytes of a vector from the higher 8-bits of two 16-bit vectors of an operand

Table 48-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
VPACK2HBM	Pack the higher two bytes of a vector from the lower 8-bits of two 16-bit vectors of an operand
VPACK2HBMF	Pack the higher two bytes of a vector from the higher 8-bits of two 16-bit vectors of an operand
VPERM	Perform byte permutation with zero or sign extension
VREP2HL	Replicate lower 16-bits into the upper 16-bits
VREP2HM	Replicate higher 16-bits to lower 16-bits
VSEXT2BHL	Vector sign extend the lower two bytes to the higher two bytes
VSEXT2BHM	Vector sign extend the higher two bytes to the lower two bytes
VSUB4B	Four way subtract
VSUB4B	Four-way 16-bit vector addition.
VSUB2H	Dual 16-bit vector subtraction
VSUBADD2H	Dual 16-bit vector subtract and add
VSUBADDS2H	Two way vector saturating subtract add
VSUBS2H	Two way vector saturating subtract 16 bits

ABSS

Function

Compute the absolute value of a 32-bit operand and saturate the result.

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT32}(\text{ABS}(c));$$

Instruction Format

op b, c

Syntax Example

ABSS<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of the input is set to 1
C		= Unchanged
V	•	= Set if src = 0x8000 0000; otherwise cleared



Note

The value of the input operands is used to set the flags.

Description

Compute the absolute value of a 32-bit operand, *c*. Saturate the result and store in operand *b*. The absolute value of 0x8000_0000 yields 0x7FFF_FFFF. The saturation flag `DSP_CTRL.SAT` is set, regardless of the `<.F>` suffix, if the instruction saturates. If the set flags suffix (`.F`) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b = c >= 0 ? c : (c == 0x8000_0000 ? 0x7fff_ffff : -c);
DSP_CTRL.SAT |= c == 0x8000_0000;
if (F) {
    STATUS32.Z = c == 0x0000_0000;
    STATUS32.N = c < 0;
    STATUS32.V = c == 0x8000_0000;
}
```

Assembly Code Example

```
ABSS r1,r2    ; Take the absolute saturated value of r2 and write
               ;result into r1
```

Syntax and Encoding

Instruction Code

ABSS<.f>	b,c	00101 bbb 00 101111 FBBB CCCCC 000101
ABSS<.f>	b,limm	00101 bbb 00 101111 FBBB 111110 000101
ABSS<.f>	0,c	00101 110 00 101111 F111 CCCCC 000101
ABSS<.f>	0,limm	00101 110 00 101111 F111 111110 000101
ABSS<.f>	b,u6	00101 bbb 01 101111 FBBB uuuuuu 000101
ABSS<.f>	0,u6	00101 110 01 101111 F111 uuuuuu 000101

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ABSSH

Function

Compute the absolute value of a 16-bit element and saturate the result.

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT16}(\text{ABS}(c.h0));$$

Instruction Format

op b, c

Syntax Example

ABSSH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of the input is set to 1
C		= Unchanged
V	•	= Set if src = 0x8000 0000; otherwise cleared



Note

The value of the input operands is used to set the flags.

Description

Compute the absolute value of a 16-bit operand *c*. Saturate and store the result in operand *b*. The absolute value of 0x8000_0000 yields 0x7FFF_FFFF. If the instruction saturates, the saturation flag DSP_CTRL.SAT is set regardless of the <.F> suffix. If the set flags suffix (.F) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
t = c & 0x0000_ffff;
b = t <= 0x7fff ? t : (t == 0x8000 ? 0x7fff : 0x10000-t);
DSP_CTRL.SAT |= t == 0x8000;
if (F) {
    STATUS32.Z = t == 0x0000;
    STATUS32.N = t >= 0x8000;
    STATUS32.V = t == 0x8000;
}
```

Assembly Code Example

```
ABSSH r1,r2    ; Take the absolute saturated value of the lower 16 bits
                ; of r2 and write result into r1
```

Syntax and Encoding

Instruction Code

ABSSH<.f>	b, c	00101 bbb 00 101111 F BBB CCCCC 000100
ABSSH<.f>	b, limm	00101 bbb 00 101111 F BBB 111110 000100
ABSSH<.f>	0, c	00101 110 00 101111 F 111 CCCCC 000100
ABSSH<.f>	0, limm	00101 110 00 101111 F 111 111110 000100
ABSSH<.f>	b, u6	00101 bbb 01 101111 F BBB uuuuuu 000100
ABSSH<.f>	0, u6	00101 110 01 101111 F 111 uuuuuu 000100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ADCS

Function

Signed addition with saturation and carry in

Extension Group

DSP basic arithmetic

Operation

$$a = \text{SAT32}(b + c + \text{STATUS32.C})$$

Instruction Format

op a, b, c

Syntax Example

ADCS<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Cleared
V	•	= Set if the result is saturated

Description

Add source operand 1 (b) and source operand 2 (c) and the carry flag (STATUS32.C); store the saturated result in the destination register, a. If the addition overflows, the result is saturated to the maximum positive value 0x7FFF_FFFF or the minimum negative value 0x8000_0000.

The saturation flag, DSP_CTRL.SAT, is set if the result of the instruction saturates, regardless of the .F suffix.

Pseudo Code

```
s = sat((sint128) b + c + STATUS32.C, 31, &a);    /* ADCS */
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
    STATUS32.C = 0;
}
```


Assembly Code Example

```
ADCS r0,r1,r2    ; Add r2 to r1 with carry and write the saturated
                  ; result to r0
```

Syntax and Encoding

Instruction Code

ADCS<.f>	a,b,c	00101bbb00100110FBBBCCCCC	AAAAAA
ADCS<.f>	a,b,u6	00101bbb01100110FBBBuuuuuu	AAAAAA
ADCS<.f>	b,b,s12	00101bbb10100110FBBBssssss	SSSSSS
ADCS<.cc><.f>	b,b,c	00101bbb11100110FBBBCCCCC0	QQQQQ
ADCS<.cc><.f>	b,b,u6	00101bbb11100110FBBBuuuuuu1	QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ADDS

Function

Signed 32-bit addition. The result is saturated.

Extension Group

DSP basic arithmetic

Operation

$$a = \text{SAT32} (b + c)$$

Instruction Format

op a, b, c

Syntax Example

ADDS<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Cleared
V	•	= Set if the output is saturated

Description

Compute the sum of operands *b* and *c*. The result is stored in the destination register, *a*. If the addition overflows, the result is saturated to the maximum positive value `0x7fff_ffff` or the minimum negative value `0x8000_0000`. The saturation flag `DSP_CTRL.SAT` is set if the result of the instruction saturates, regardless of the `.F` suffix. If the set flags suffix (`.F`) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
s = sat((sint128) b + c, 31, &a);          /* ADDS */
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
    STATUS32.C = 0;
}
```

Assembly Code Example

```
ADDS r0,r1,r2    ; Add contents of r1 with r2 and write result into r0
                  ; while saturating
```

Syntax and Encoding

Instruction Code

ADDS<.f>	a, b, c	00101bbb00000110FBBBCCCCC	AAAAAA
ADDS<.f>	a, b, u6	00101bbb01000110FBBBuuuuuu	AAAAAA
ADDS<.f>	b, b, s12	00101bbb10000110FBBBssssss	SSSSSS
ADDS<.cc><.f>	b, b, c	00101bbb11000110FBBBCCCCC	0QQQQQ
ADDS<.cc><.f>	b, b, u6	00101bbb11000110FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ASLACC

Function

Perform an arithmetic shift of the accumulator.

Extension Group

DSP Accumulator

Operation

```
switch (c.b1) {
0: accwide = accwide << c.b0;
1: acclo = acclo << c.b0;
2: acchi = acchi << c.b0;
3: acchi = acchi << c.b2; acclo = acclo << c.b0;
}
```

Instruction Format

op c

Syntax Example

ASLACC c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an arithmetic left shift of the accumulator, including guard bits. The `c` operand consists of two fields: the shift amount (`c.b0`) and the accumulator select (`c.b1`). The shift amount is a signed integer; a negative value effectively shifts the accumulator right, without rounding.

Using the loop-counter (LP_COUNT/r60) as a source operand to the ASLACC instruction results in an illegal instruction exception.

If, `dsp_accshift==full`, the shift amount is saturated to a range of $-72 \leq \text{shamt} \leq +72$; if `dsp_accshift==limited`, the shift amount is saturated to a range of $-8 \leq \text{shamt} \leq +8$.

Operand Attributes

Operand c.b1	Description
0	selects the wide accumulator
1	selects the low accumulator
2	selects the high accumulator
3	selects dual hi/lo accumulator output; move c.b2 into the high accumulator, and move c.b0 into the low accumulator

Pseudo Code

```

if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)                                     // ASLACC
    shamt = c.b0 < -72 ? -72 : (c.b0 > 72 ? 72 : c.b0);
else
    shamt = c.b0 < -8 ? -8 : (c.b0 > 8 ? 8 : c.b0);
switch (c.b1 & 3) {
    case 0:
        if (DSP_CTRL.GE)
            accwide = ACC0_GHI.GUARD << 64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        accwide = (shamt >= 0) ? accwide << shamt : accwide >> -shamt;
        if (DSP_CTRL.GE)
            ACC0_GHI.GUARD = accwide >> 64 && 0xff;
            ACC0_HI = accwide >> 32 && 0xffff_ffff;
            ACC0_LO = accwide && 0xffff_ffff;
            break;
    case 1:
        if (DSP_CTRL.GE)
            acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
        else
            acclo = ACC0_LO;
        acclo = (shamt >= 0) ? acclo << shamt : acclo >> -shamt;
        if (DSP_CTRL.GE)
            ACC0_GLO.GUARD = acclo >> 32 && 0xff;
            ACC0_LO = acclo && 0xffff_ffff;
            break;
    case 2:
        if (DSP_CTRL.GE)
            acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        else
            acchi = ACC0_HI;
        acchi = (shamt >= 0) ? acchi << shamt : acchi >> -shamt;
        if (DSP_CTRL.GE)
            ACC0_GHI.GUARD = acchi >> 32 && 0xff;
            ACC0_HI = acchi && 0xffff_ffff;
            break;
}

```

```

case 3:
    shamtl = sext(c.b0);
    shamth = sext(c.b2);
    if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)
        shamtl = shamtl < -72 ? -72 : (shamtl > 72 ? 72 : shamtl);
        shamth = shamth < -72 ? -72 : (shamth > 72 ? 72 : shamth);
    else
        shamtl = shamtl < -8 ? -8 : (shamtl > 8 ? 8 : shamtl);
        shamth = shamth < -8 ? -8 : (shamth > 8 ? 8 : shamth);
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
    else
        acchi = ACC0_HI;
        acclo = ACC0_LO;
    acchi = (shamth >= 0) ? acchi << shamth : acchi >> -shamth;
    acclo = (shamtl >= 0) ? acclo << shamtl : acclo >> -shamtl;
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = acchi >> 32 && 0xff;
        ACC0_GLO.GUARD = acclo >> 32 && 0xff;
    ACC0_HI = acchi && 0xffff_ffff;
    ACC0_LO = acclo && 0xffff_ffff;
    break;
}

```

Assembly Code Example

```
ASLACC r2          ; Shift left the accumulator based on fields in r2
```

Syntax and Encoding

Instruction Code		
ASLACC	c	00101000001011110000CCCCC111111
ASLACC	u6	00101000011011110000uuuuuu111111

ASLS

Function

Perform an arithmetic shift left operation on an operand. The shift amount is specified in the second operand. Store the saturated result in the destination register.

Extension Group

DSP basic arithmetic

Operation

$$a = c >= 0 ? \text{SAT32}(b \ll c) : b \gg -c$$

Instruction Format

op a, b, c

Syntax Example

ASLS<.f> a, b, c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if the result is saturated

Description

If operand *c* is positive, perform an arithmetic shift left on operand *b* by the shift amount is specified by in the *c* operand. Return the saturated result. If operand *c* is negative, perform a shift right by $-c$ bits. The left-shift operation saturates if any of the bits shifted out is unequal to the original sign bit or if the result sign bit is different from the original sign bit.

The shift amount is saturated to $[-31, 32]$.

The saturation flag `DSP_CTRL.SAT` is set if the result of the instruction saturates regardless of the `.F` suffix. Other flag updates only occur if the set flags suffix (`.F`) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

if (c >= 0) {
    c = c > 32 ? 32 : c;
    s = sat((sint128) b << c, 31, &a);
} else {
    s = 0;
    a = b >> (c < -31 ? 31 : -c);
}
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
}

```

Assembly Code Example

```

ASLS r0,r1,r2      ; Arithmetic shift left r1 by r2 bit positions and write
                   ;result into r0 while saturating

```

Syntax and Encoding

		Instruction Code
ASLS<.f>	a,b,c	00101 bbb 00 001010 F BBB CCCCC AAAAAA
ASLS<.f>	a,b,u6	00101 bbb 01 001010 F BBB uuuuuu AAAAAA
ASLS<.f>	b,b,s12	00101 bbb 10 001010 F BBB ssssss SSSSSS
ASLS<.f><.cc>	b,b,c	00101 bbb 11 001010 F BBB CCCCC 0QQQQQ
ASLS<.f><.cc>	b,b,u6	00101 bbb 11 001010 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ASLSACC

Function

Perform an arithmetic shift left operation on the accumulator. Store the saturated result.

Extension Group

DSP Accumulator

Operation

```
if (DSP_CTRL.GE) {
    switch (c.b1) {
        0: accwide = SAT72(accwide << c.b0);
        1: acclo = SAT40(acclo << c.b0);
        2: acchi = SAT40(acchi << c.b0);
        3: acchi = SAT40(acchi << c.b2); acclo = SAT40(acclo << c.b0);
    } else {
        switch (c.b1) {
            0: accwide = SAT64(accwide << c.b0);
            1: acclo = SAT32(acclo << c.b0);
            2: acchi = SAT32(acchi << c.b0);
            3: acchi = SAT32(acchi << c.b2); acclo = SAT32(acclo << c.b0);
        }
    }
}
```

Instruction Format

op c

Syntax Example

ASLSACC c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an arithmetic shift left operation on the accumulator including the guard bits. Store the saturated result. The c operand consists of two fields: the shift amount and the accumulator select.

The shift amount is a signed integer; a negative value effectively shifts the accumulator right, with saturation and without rounding. The sticky flag, `DSP_CTRL.SAT`, is set if the result saturates.

If, `dsp_accshift==full`, the shift amount is saturated to a range of $-72 \leq \text{shamt} \leq +72$; if `dsp_accshift==limited`, the shift amount is saturated to a range of $-8 \leq \text{shamt} \leq +8$.

Using the loop-counter (`LP_COUNT/r60`) as a source operand to the `ASLSACC` instruction results in an illegal instruction exception.

Operand Attributes

Operand c.b1	Description
0	selects the wide accumulator
1	selects the low accumulator
2	selects the high accumulator
3	selects dual hi/lo accumulator output; move c.b2 into the high accumulator, and move c.b0 into the low accumulator

Pseudo Code

```

if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)                                /* ASLSACC
    shamt = c.b0 < -72 ? -72 : (c.b0 > 72 ? 72 : c.b0);                */
else
    shamt = c.b0 < -8 ? -8 : (c.b0 > 8 ? 8 : c.b0);
switch (c.b1 & 3) {
    case 0:
        if (DSP_CTRL.GE)
            accwide = ACC0_GHI.GUARD << 64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            msb = 71;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            msb = 63;
        if (shamt >= 0)
            s = sat(accwide<<shamt, msb, &accwide); // requires 256b arith
        else
            s = 0;
            accwide = accwide >> -shamt;
        DSP_CTRL.SAT |= s;
        if (DSP_CTRL.GE)
            ACC0_GHI.GUARD = accwide >> 64 && 0xff;
            ACC0_HI = accwide >> 32 && 0xffff_ffff;
            ACC0_LO = accwide && 0xffff_ffff;
            break;
    case 1:
        if (DSP_CTRL.GE)
            acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
            msb = 39;
        else
            acclo = ACC0_LO;
            msb = 31;
        if (shamt >= 0)
            s = sat(acclo<<shamt, msb, &acclo); // requires 256b arith
        else
            s = 0;
            acclo = acclo >> -shamt;
        DSP_CTRL.SAT |= s;
        if (DSP_CTRL.GE)
            ACC0_GLO.GUARD = acclo >> 32 && 0xff;
            ACC0_LO = acclo && 0xffff_ffff;
            break;

```

```

case 2:
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        msb = 39;
    else
        acchi = ACC0_HI;
        msb = 31;
    if (shamt >= 0)
        s = sat(acchi<<shamt, msb, &acchi); // requires 256b arith
    else
        s = 0;
        acchi = acchi >> -shamt;
    DSP_CTRL.SAT |= s;
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = acchi >> 32 && 0xff;
    ACC0_HI = acchi && 0xffff_ffff;
    break;
case 3:
    shamtl = sext(c.b0);
    shamth = sext(c.b2);
    if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)
        shamtl = shamtl < -72 ? -72 : (shamtl > 72 ? 72 : shamtl);
        shamth = shamth < -72 ? -72 : (shamth > 72 ? 72 : shamth);
    else
        shamtl = shamtl < -8 ? -8 : (shamtl > 8 ? 8 : shamtl);
        shamth = shamth < -8 ? -8 : (shamth > 8 ? 8 : shamth);
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
        msb = 39;
    else
        acchi = ACC0_HI;
        acclo = ACC0_LO;
        msb = 31;
    if (shamtl >= 0)
        sl = sat(acclo<<shamtl, msb, &acclo); // requires 256b arith
    else
        sl = 0;
        acclo = acclo >> -shamtl;
    if (shamth >= 0)
        sh = sat(acchi<<shamth, msb, &acchi); // requires 256b arith
    else
        sh = 0;
        acchi = acchi >> -shamth;
    DSP_CTRL.SAT |= sl || sh;
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = acchi >> 32 && 0xff;
        ACC0_GLO.GUARD = acclo >> 32 && 0xff;
    ACC0_HI = acchi && 0xffff_ffff;
    ACC0_LO = acclo && 0xffff_ffff;
    break;
}

```

```

case 3:
    shamtl = sext(c.b0);
    shamth = sext(c.b2);
    shamtl = shamtl < -64 ? -64 : (shamtl > 63 ? 63 : shamtl);
    shamth = shamth < -64 ? -64 : (shamth > 63 ? 63 : shamth);
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
        msb = 39;
    else
        acchi = ACC0_HI;
        acclo = ACC0_LO;
        msb = 31;
    if (shamtl >= 0)
        sl = sat(acclo<<shamtl, msb, &acclo); // requires 256b arith
    else
        sl = 0;
        acclo = acclo >> -shamtl;
    if (shamth >= 0)
        sh = sat(acchi<<shamth, msb, &acchi); // requires 256b arith
    else
        sh = 0;
        acchi = acchi >> -shamth;
    DSP_CTRL.SAT |= sl || sh;
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = acchi >> 32 && 0xff;
        ACC0_GLO.GUARD = acclo >> 32 && 0xff;
    ACC0_HI = acchi && 0xffff_ffff;
    ACC0_LO = acclo && 0xffff_ffff;
    break;
}

```

Assembly Code Example

```

ASLSACC r2      ; Shift left the accumulator and saturate based on fields
                 ;in r2

```

Syntax and Encoding

Instruction Code

ASLSACC	c	00101001001011110000CCCCC111111
ASLSACC	u6	00101001011011110000uuuuuu111111

ASRS

Function

Perform an arithmetic shift right operation on an operand. The shift amount is specified in the second operand. Store the saturated result in the destination register.

Extension Group

DSP basic arithmetic

Operation

$$a = c \gg 0 ? b \gg c : \text{SAT32}(b \ll -c) ;$$

Instruction Format

op a, b, c

Syntax Example

ASRS<.f> a, b, c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if the result is saturated

Description

Perform an arithmetic shift right operation on an operand. The shift amount is specified in the *c* operand. Store the saturated result in the destination register. If operand *c* is negative, perform an arithmetic shift left—shift amount is specified in the *c* operand and return the saturated result. The left shift saturates if any of the bits shifted out is unequal to the original sign bit or if the result sign bit is different from the original sign bit.

The shift amount is saturated to [-32,31].

The saturation flag `DSP_CTRL.SAT` is set if the result of the instruction saturates regardless of the *.F* suffix. Other flag updates only occur if the set flags suffix (*.F*) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

if (c >= 0) {
    s = 0;
    a = b >> (c > 31 ? 31 : c);
} else {
    c = c < -32 ? -32 : c;
    s = sat((sint128) b << -c, 31, &a);
}
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
}

```

Assembly Code Example

```

ASRS r0,r1,r2    ; Arithmetic shift right r1 by r2 bit positions and write
                  ;result into r0 while saturating

```

Syntax and Encoding

Instruction Code

ASRS<.f>	a,b,c	00101 bbb 00001011 FBBB CCCCC AAAAAA
ASRS<.f>	a,b,u6	00101 bbb 01001011 FBBB uuuuuu AAAAAA
ASRS<.f>	b,b,s12	00101 bbb 10001011 FBBB ssssss SSSSSS
ASRS<.cc><.f>	b,b,c	00101 bbb 11001011 FBBB CCCCC 0QQQQQ
ASRS<.cc><.f>	b,b,u6	00101 bbb 11001011 FBBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ASRSR

Function

Perform an arithmetic shift right operation on an operand. The shift amount is specified in the second operand. Store the rounded and saturated result in the destination register.

Extension Group

DSP basic arithmetic

Operation

$$a = c \gg 0 ? \text{RND32}(b \gg c) : \text{SAT32}(b \ll -c) ;$$

Instruction Format

op a, b, c

Syntax Example

ASRSR<.f> a, b, c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if the result is saturated

Description

Perform an arithmetic shift right operation on an operand. The shift amount is specified in the *c* operand. Store the rounded and saturated result in the destination register. If operand *c* is negative, perform an arithmetic shift left by the shift amount is specified in the *c* operand, and return the saturated result. The left shift saturates if any of the bits shifted out is unequal to the original sign bit or if the result sign bit is different from the original sign bit. The rounding mode is based on the `DSP_CTRL.RM` bits.

The shift amount is saturated to [-32,32].

The saturation flag `DSP_CTRL.SAT` is set if the result of the instruction saturates, regardless of the *.F* suffix. Other flag updates only occur if the set flags suffix (*.F*) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

if (c >= 0) {
    s = 0;
    c = c > 32 ? 32 : c;
    a = round(b, c) >> c;
} else {
    c = c < -32 ? -32 : c;
    s = sat((sint128) b << -c, 31, &a);
}
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
}

```

Assembly Code Example

```

ASRSR r0,r1,r2    ; Arithmetic shift right r1 by r2 bit positions and write
                  ;result into r0 while saturating and rounding

```

Syntax and Encoding

Instruction Code

ASRSR<.f>	a,b,c	00101bbb00001100FBBBCCCCC
ASRSR<.f>	a,b,u6	00101bbb01001100FBBBuuuuuuAAAAAA
ASRSR<.f>	b,b,s12	00101bbb10001100FBBBssssssSSSSSS
ASRSR<.cc><.f>	b,b,c	00101bbb11001100FBBBCCCCC0QQQQQ
ASRSR<.cc><.f>	b,b,u6	00101bbb11001100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CBFLYHF0R

Function

Compute the first result of a fast Fourier transform (FFT) butterfly operation. Store the rounded and saturated result.

Extension Group

DSP Accumulator

Operation

```

accr = bfly.real + b.r * c.r - b.i * c.i;
acci = bfly.imag + b.i * c.r + b.r * c.i;
a.r  = sat(accr);
a.i  = sat(acci);

```

Instruction Format

op a, b, c

Syntax Example

CBFLYHF0R a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compute the first rounded result of an FFT Butterfly operation, based on the twiddle factor in the accumulator: $A' = A + T * B$.

Store the result in the accumulator. The instruction does set any STATUS32 flags. Optionally, the result can be scaled. The scaling is controlled by the auxiliary register, `DSP_FFT_CTRL`. SC bit. The third butterfly operand is from the auxiliary register, `DSP_BFLY0` register.

This instruction is sensitive to the processor endian mode configuration parameter. In little-endian mode, the low accumulator stores the real part of the result and in the big-endian mode, the low accumulator stores the imaginary part of the result.

An FFT butterfly performs the following operation in which all variables are complex numbers:

```

A1 = A0 + T * B0
B1 = A0 - T * B0 = 2 * A0 - A1

```

The butterfly computations require three inputs and write two outputs. The ARCV2 architecture only supports two explicit input operands to an instruction which is why the A0 input to an FFT butterfly is taken from an auxiliary register (DSP_BFLY0).

```
CBFLYHF0R A1,T,B0      ;compute A1 result using T and B0 explicit input
                        ;operands and DSP_BFLY0 AUX implicit input operand

CBFLYH1FR B1,A0        ; compute B1 result using T A1 and A0 from previous
                        ;iteration and set next A0 value in DSP_BFLY0 AUX
                        ;register.
```

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// Accumulator stores A1 result
if (BYTE_ORDER == little_endian) {
    ACC0_GHI.V = sat((DSP_BFLY0.IMAG<<15) + b.h1 * c.h0 + b.h0 * c.h1, 31,
&ACC0_HI);
    ACC0_GLO.V = sat((DSP_BFLY0.REAL<<15) + b.h0 * c.h0 - b.h1 * c.h1, 31,
&ACC0_LO);
} else {
    ACC0_GHI.V = sat((DSP_BFLY0.REAL<<15) + b.h1 * c.h1 - b.h0 * c.h0, 31,
&ACC0_HI);
    ACC0_GLO.V = sat((DSP_BFLY0.IMAG<<15) + b.h1 * c.h0 + b.h0 * c.h1, 31,
&ACC0_LO);
}
if (DSP_FFT_CTRL.SC) {
    s1 = sat(round(ACC0_HI, 16) >> 16, 15, &a.h1);
    sh = sat(round(ACC0_LO, 16) >> 16, 15, &a.h0);
} else {
    s1 = sat(round(ACC0_HI, 15) >> 15, 15, &a.h1);
    sh = sat(round(ACC0_LO, 15) >> 15, 15, &a.h0);
}
ACC0_GHI.Z = ACC0_HI == 0;
ACC0_GHI.N = ACC0_HI < 0;
ACC0_GHI.G = 0;
ACC0_GLO.Z = ACC0_LO == 0;
ACC0_GLO.N = ACC0_LO < 0;
ACC0_GLO.G = 0;
DSP_CTRL.SAT |= ACC0_GHI.V || ACC0_GLO.V || s1 || sh;
```

Assembly Code Example

```
LD.AB %r5, [%r0, 4]      ; load A
CBFLYHF1R 0, %r5         ; set A into accumulator, ignore return value
LP lpend
LD.AB %r6, [%r1, 4]      ; load B
LD.AB %r6, [%r1, 4]      ; load twiddle factor
LD.AB %r7, [%r2, 4]      ; load B
CBLFYHF0R %r8, %r6, %r7  ; compute A1 = A0 + T * B0
LD.AB %r5, [%r0, 4]      ; next load A
CBFLYHF1R %r9, %r5       ; compute B1 = A0 - T * B0, set next A0 in register
ST.AB %r8, [%r3, 4]      ; write output A1
ST.AB %r9, [%r4, 4]      ; write output B1
lpend:
```

C Code Example

The butterfly computations require three inputs and write two outputs. The ARCV2 ISA supports only two explicit input operands to an instruction. Therefore the A0 input to an FFT butterfly is taken from an auxiliary register (DSP_BFLY0); the A0 value is updated by software pipelining the butterfly computation.

The following example illustrates how these two instructions can be used

```
cbflyhf1r(*ain++);          // first A input, ignore first
for (i = 0; i < max; i++)   // return value
{
    *aout++ = cbflyhf0r(*bin++, *twiddle++); // B & twiddle input, A output
    *bout++ = cbflyhf1r(*ain++);           // B output, next A input
}
```

Syntax and Encoding

Instruction Code		
CBFLYHF0R	a, b, c	00110bbb000110111BBBCCCCCAAAAAA
CBFLYHF0R	a, b, u6	00110bbb010110111BBBuuuuuuAAAAAA
CBFLYHF0R	b, b, s12	00110bbb100110111BBBssssssSSSSSS
CBFLYHF0R<.cc>	b, b, c	00110bbb110110111BBBCCCCC0QQQQQ
CBFLYHF0R<.cc>	b, b, u6	00110bbb110110111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

`SUB.F 0, R0, R1`

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CBFLYHF1R

Function

Compute the second result of a fast Fourier transform butterfly operation. Store the rounded and saturated result.

Extension Group

DSP Accumulator

Operation

```
accr = 2 * bfly.real - accr;
acci = 2 * bfly.imag - acci;
b.i = sat(acci); b.r = sat(accr);
bfly = c
```

Instruction Format

op b, c

Syntax Example

CBFLYHF1R b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compute the rounded second result of an FFT Butterfly operation, based on the twiddle factor in the accumulator: $B1 = A0 - T * B = 2 * A0 - A1$.

Store the next A0 in the accumulator. The instruction does set any STATUS32 flags. Optionally, the result can be scaled. The scaling is controlled by the auxiliary register, `DSP_FFT_CTRL.SC` bit. The third butterfly operand is from the auxiliary register, `DSP_BFLY0` register.

This instruction is sensitive to the processor endian mode configuration parameter. In the little-endian mode, the low accumulator stores the real part of the result and in the big-endian mode, the low accumulator stores the imaginary part of the result.

An FFT butterfly performs the following operation in which all variables are complex numbers:

$$A1 = A0 + T * B0$$

$$B1 = A0 - T * B0 = 2 * A0 - A1$$

The butterfly computations require three inputs and write two outputs. The ARCV2 architecture only supports two explicit input operands to an instruction which is why the A0 input to an FFT butterfly is taken from an auxiliary register (DSP_BFLY0).

```
CBFLYHF0R A1,T,B0    ;compute A1 result using T and B0 explicit input
                    ;operands and DSP_BFLY0 AUX implicit input operand

CBFLYH1FR B1,A0      ;compute B1 result using T A1 and A0 from previous
                    ;iteration and set next A0 value in DSP_BFLY0 AUX
                    ;register.
```

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// Compute 2*A0-A1                                     /* CBFLYHF1R*/
if (BYTE_ORDER == little_endian) {
    ACC0_GHI.V = sat((DSP_BFLY0.IMAG << 16) - ACC0_HI, 31, &ACC0_HI);
    ACC0_GLO.V = sat((DSP_BFLY0.REAL << 16) - ACC0_LO, 31, &ACC0_LO);
} else {
    ACC0_GHI.V = sat((DSP_BFLY0.REAL << 16) - ACC0_HI, 31, &ACC0_HI);
    ACC0_GLO.V = sat((DSP_BFLY0.IMAG << 16) - ACC0_LO, 31, &ACC0_LO);
}
if (DSP_FFT_CTRL.SC) {
    sh = sat(round(ACC0_HI, 16) >> 16, 15, &b.h1);
    sl = sat(round(ACC0_LO, 16) >> 16, 15, &b.h0);
} else {
    sh = sat(round(ACC0_HI, 15) >> 15, 15, &b.h1);
    sl = sat(round(ACC0_LO, 15) >> 15, 15, &b.h0);
}
ACC0_GHI.Z = ACC0_HI == 0;
ACC0_GHI.N = ACC0_HI < 0;
ACC0_GHI.G = 0;
ACC0_GLO.Z = ACC0_LO == 0;
ACC0_GLO.N = ACC0_LO < 0;
ACC0_GLO.G = 0;
// set next A0
DSP_BFLY = c;
DSP_CTRL.SAT |= ACC0_GHI.V || ACC0_GLO.V || sl || sh;
```


Assembly Code Example

```
LD.AB %r5, [%r0, 4]      ; load A
CBFLYHF1R 0, %r5         ; set A into accumulator, ignore return value
LP lpend
LD.AB %r6, [%r1, 4]      ; load B
LD.AB %r6, [%r1, 4]      ; load twiddle factor
LD.AB %r7, [%r2, 4]      ; load B
CBFLYHF0R %r8, %r6, %r7  ; compute A1 = A0 + T * B0
LD.AB %r5, [%r0, 4]      ; next load A
CBFLYHF1R %r9, %r5       ; compute B1 = A0 - T * B0, set next A0 in register
ST.AB %r8, [%r3, 4]      ; write output A1
ST.AB %r9, [%r4, 4]      ; write output B1
lpend:
```

C Code Example

The butterfly computations require three inputs and write two outputs. The ARCV2 ISA supports only two explicit input operands to an instruction. Therefore the A0 input to an FFT butterfly is taken from an auxiliary register (DSP_BFLY0); the A0 value is updated by software pipelining the butterfly computation.

The following example illustrates how these two instructions can be used

```
cbflyhf1r(*ain++);          // first A input, ignore first
for (i = 0; i < max; i++)   return value
{
    *aout++ = cbflyhf0r(*bin++,          // B & twiddle input, A output
    *twiddle++);              // B output, next A input
    *bout++ = cbflyhf1r(*ain++);
}
```

Syntax and Encoding

Instruction Code

CBFLYHF1R	b, c	00110 bbb 00 101111 0 BBB CCCCC 011001
CBFLYHF1R	b, limm	00110 bbb 00 101111 0 BBB 111110 011001
CBFLYHF1R	0, c	00110 110 00 101111 0 111 CCCCC 011001
CBFLYHF1R	0, limm	00110 110 00 101111 0 111 111110 011001
CBFLYHF1R	b, u6	00110 bbb 01 101111 0 BBB uuuuuu 011001
CBFLYHF1R	0, u6	00110 110 01 101111 0 111 uuuuuu 011001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMACCHFR

Function

Multiply and accumulate two signed fractional complex conjugate operands. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16(RND16((acci + b.i*c.r - b.r*c.i)<<1));
a.r = SAT16(RND16((accr + b.r*c.r + b.i*c.i)<<1));
```

Instruction Format

op a,b,c

Syntax Example

CMACCHFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply and accumulate the two signed fractional complex conjugate operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        if (BYTE_ORDER == little_endian) {
            addr0 = b.h0 * c.h0 << 1;
            addr1 = b.h1 * c.h1 << 1;
            addi0 = b.h1 * c.h0 << 1;
            addi1 = b.h0 * -c.h1 << 1;
        } else {
            addr0 = b.h1 * -c.h0 << 1;
            addr1 = b.h0 * c.h1 << 1;
            addi0 = b.h0 * c.h0 << 1;
            addi1 = b.h1 * c.h1 << 1;
        }
        acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &alo);
        acchi           = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &ahi);
    else
        // guard bits disabled
        if (BYTE_ORDER == little_endian) {
            s |= sat((b.h0 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h1 * c.h1 << 1), 31, &addr1);
            s |= sat((b.h1 * c.h0 << 1), 31, &addi0);
            s |= sat((b.h0 * -c.h1 << 1), 31, &addi1);
            addi1 = -addi1;
        } else {
            s |= sat((b.h1 * -c.h0 << 1), 31, &addr0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addr1);
            s |= sat((b.h0 * c.h0 << 1), 31, &addi0);
            s |= sat((b.h1 * c.h1 << 1), 31, &addi1);
            addr0 = -addr0;
        }
        acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &alo);

```

```

acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    if (BYTE_ORDER == little_endian) {
        addr0 = b.h0 * c.h0;
        addr1 = b.h1 * c.h1;
        addi0 = b.h1 * c.h0;
        addi1 = b.h0 * -c.h1;
    } else {
        addr0 = b.h1 * -c.h0;
        addr1 = b.h0 * c.h1;
        addi0 = b.h0 * c.h0;
        addi1 = b.h1 * c.h1;
    }
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1,16), 31, &alo);
        acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);
    DSP_CTRL.SAT |= s | ACC0_GLO.V | ACC0_GHI.V;
    a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMACCHFR r0,r1,r2    ;Signed complex conjugated multiply-accumulate
                     ; two complex vectors r1 and r2 and
                     ; return the saturated and rounded result in r0
```

Syntax and Encoding

Instruction Code		
CMACCHFR	a, b, c	00110bbb000010011BBBCCCCCAAAAA
CMACCHFR	a, b, u6	00110bbb010010011BBBuuuuuuAAAAA
CMACCHFR	b, b, s12	00110bbb100010011BBBssssssSSSSSS
CMACCHFR<.cc>	b, b, c	00110bbb110010011BBBCCCCC0QQQQQ
CMACCHFR<.cc>	b, b, u6	00110bbb110010011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMACCHNFR

Function

Multiply and accumulate two signed fractional complex conjugate operands. The result is rounded and saturated but is returned without a fractional shift.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16((acci + b.i*c.r - b.r*c.i));
a.r = SAT16((accr + b.r*c.r + b.i*c.i));
```

Instruction Format

op a,b,c

Syntax Example

CMACCHNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply and accumulate two signed fractional complex conjugate operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector without a fractional shift in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0;
    addr1 = b.h1 * c.h1;
    addi0 = b.h1 * c.h0;
    addi1 = b.h0 * -c.h1;
} else {
    addr0 = b.h1 * -c.h0;
    addr1 = b.h0 * c.h1;
    addi0 = b.h0 * c.h0;
    addi1 = b.h1 * c.h1;
}
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);
```


Assembly Code Example

```
CMACCHNFR r0,r1,r2    ; Signed complex multiply-accumulate
                      ; two complex vectors r1 and r2 and
                      ; return result in r0 with saturation and rounding
                      ; and returned without a fractional shift
```

Syntax and Encoding

Instruction Code		
CMACCHNFR	a, b, c	00110bbb000010001BBBCCCCCAAAAAA
CMACCHNFR	a, b, u6	00110bbb010010001BBBuuuuuuAAAAAA
CMACCHNFR	b, b, s12	00110bbb100010001BBBssssssSSSSSS
CMACCHNFR<.cc>	b, b, c	00110bbb110010001BBBCCCCC0QQQQQ
CMACCHNFR<.cc>	b, b, u6	00110bbb110010001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMACHFR

Function

Multiply and accumulate two signed fractional complex operands. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16(RND16((acci + b.i*c.r + b.r*c.i)<<1));
a.r = SAT16(RND16((accr + b.r*c.r - b.i*c.i)<<1));
```

Instruction Format

op a,b,c

Syntax Example

CMACHFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply and accumulate the two signed fractional complex operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        if (BYTE_ORDER == little_endian) {
            addr0 = b.h0 * c.h0 << 1;
            addr1 = b.h1 * -c.h1 << 1;
            addi0 = b.h1 * c.h0 << 1;
            addi1 = b.h0 * c.h1 << 1;
        } else {
            addr0 = b.h1 * c.h0 << 1;
            addr1 = b.h0 * c.h1 << 1;
            addi0 = b.h0 * -c.h0 << 1;
            addi1 = b.h1 * c.h1 << 1;
        }
        acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z,
&ACC0_GLO.N, &ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        s                 |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        s                 |= sat(round(acclo, 16), 31, &ahi);
    else
        // guard bits disabled
        if (BYTE_ORDER == little_endian) {
            s |= sat((b.h0 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h1 * -c.h1 << 1), 31, &addr1);
            s |= sat((b.h1 * c.h0 << 1), 31, &addi0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addi1);
            addr1 = -addr1;
        } else {
            s |= sat((b.h1 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addr1);
            s |= sat((b.h0 * -c.h0 << 1), 31, &addi0);
            s |= sat((b.h1 * c.h1 << 1), 31, &addi1);
            addi0 = -addi0;
        }
        acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z,
&ACC0_GLO.N, &ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        s                 |= sat(round(acclo, 16), 31, &alo);

```

```

acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
  ACC0_GHI.G      = 0;
  ACC0_HI         = acchi & 0xffff_ffff;
  s               |= sat(round(acchi, 16), 31, &ahi);
else
  // post-accumulate fractional shift mode
  if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0;
    addr1 = b.h1 * -c.h1;
    addi0 = b.h1 * c.h0;
    addi1 = b.h0 * c.h1;
  } else {
    addr0 = b.h1 * c.h0;
    addr1 = b.h0 * c.h1;
    addi0 = b.h0 * -c.h0;
    addi1 = b.h1 * c.h1;
  }
  if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z,
&ACC0_GLO.N, &ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi<<1, 16), 31, &ahi);
  else
    // guard bits are disabled
    acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z,
&ACC0_GLO.N, &ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi<<1, 16), 31, &ahi);
  DSP_CTRL.SAT |= s | ACC0_GLO.V | ACC0_GHI.V;
  a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMACHFR r0,r1,r2      ;Signed complex conjugated multiply-accumulate
                       ; two complex vectors r1 and r2 and
                       ; return the saturated and rounded result in r0
```

Syntax and Encoding

Instruction Code		
CMACHFR	a, b, c	00110bbb000001111BBBCCCCCAAAAAA
CMACHFR	a, b, u6	00110bbb010001111BBBuuuuuuAAAAAA
CMACHFR	b, b, s12	00110bbb100001111BBBssssssSSSSSS
CMACHFR<.cc>	b, b, c	00110bbb110001111BBBCCCCC0QQQQQ
CMACHFR<.cc>	b, b, u6	00110bbb110001111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMACHNFR

Function

Multiply and accumulate two signed fractional complex operands. The result is rounded and saturated, and is returned without a fractional shift.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16((acci + b.i*c.r + b.r*c.i));
a.r = SAT16((accr + b.r*c.r - b.i*c.i));
```

Instruction Format

op a,b,c

Syntax Example

CMACHNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply and accumulate two signed fractional complex operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector without any fractional shift in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0;
    addr1 = b.h1 * -c.h1;
    addi0 = b.h1 * c.h0;
    addi1 = b.h0 * c.h1;
} else {
    addr0 = b.h1 * c.h0;
    addr1 = b.h0 * c.h1;
    addi0 = b.h0 * -c.h0;
    addi1 = b.h1 * c.h1;
}
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
    &ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
    &ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
    &ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
    &ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V ;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMACHNFR r0,r1,r2    ;Signed complex multiply-accumulate
                     ; two complex vectors r1 and r2 and
                     ; return result in r0 with saturation and rounding
                     ;without any fractional shift
```

Syntax and Encoding

Instruction Code		
CMACHNFR	a, b, c	00110bbb000001101BBBCCCCCAAAAAA
CMACHNFR	a, b, u6	00110bbb010001101BBBuuuuuuAAAAAA
CMACHNFR	b, b, s12	00110bbb100001101BBBssssssSSSSSS
CMACHNFR<.cc>	b, b, c	00110bbb110001101BBBCCCCC0QQQQQ
CMACHNFR<.cc>	b, b, u6	00110bbb110001101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYCHFR

Function

Multiply two signed fractional complex conjugate operands. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.h1 = SAT16(RND16((b.i*c.r - b.r*c.i)<<1));
a.h0 = SAT16(RND16((b.r*c.r + b.i*c.i)<<1));
```

Instruction Format

op a,b,c

Syntax Example

CMPYCHFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply two signed fractional complex conjugate operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
if (DSP_CTRL.PA)
// pre-accumulate fractional shift mode
if (DSP_CTRL.GE)
// guard bits are enabled
if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0 << 1;
    addr1 = b.h1 * c.h1 << 1;
    addi0 = b.h1 * c.h0 << 1;
    addi1 = b.h0 * -c.h1 << 1;
} else {
    addr0 = b.h1 * -c.h0 << 1;
    addr1 = b.h0 * c.h1 << 1;
    addi0 = b.h0 * c.h0 << 1;
    addi1 = b.h1 * c.h1 << 1;
}
acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
ACC0_LO         = acclo & 0xffff_ffff;
s               |= sat(round(acclo, 16), 31, &alo);
acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
ACC0_HI         = acchi & 0xffff_ffff;
S               |= sat(round(acclo, 16), 31, &ahi);
else
// guard bits disabled
if (BYTE_ORDER == little_endian) {
    s |= sat((b.h0 * c.h0 << 1), 31, &addr0);
    s |= sat((b.h1 * c.h1 << 1), 31, &addr1);
    s |= sat((b.h1 * c.h0 << 1), 31, &addi0);
    s |= sat((b.h0 * -c.h1 << 1), 31, &addi1);
    addi1 = -addi1;
} else {
    s |= sat((b.h1 * -c.h0 << 1), 31, &addr0);
    s |= sat((b.h0 * c.h1 << 1), 31, &addr1);
    s |= sat((b.h0 * c.h0 << 1), 31, &addi0);
    s |= sat((b.h1 * c.h1 << 1), 31, &addi1);
    addr0 = -addr0;
}
acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
ACC0_GLO.G      = 0;
ACC0_LO         = acclo & 0xffff_ffff;
s               |= sat(round(acclo, 16), 31, &alo);

```

```

    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    if (BYTE_ORDER == little_endian) {
        addr0 = b.h0 * c.h0;
        addr1 = b.h1 * c.h1;
        addi0 = b.h1 * c.h0;
        addi1 = b.h0 * -c.h1;
    } else {
        addr0 = b.h1 * -c.h0;
        addr1 = b.h0 * c.h1;
        addi0 = b.h0 * c.h0;
        addi1 = b.h1 * c.h1;
    }
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1,16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 00
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);
    DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V;
    a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMPYCHFR r0,r1,r2    ;Signed complex conjugated multiply
                    ; two complex vectors r1 and r2 and
                    ; return result in r0 with saturation and rounding
```

Syntax and Encoding

Instruction Code		
CMPYCHFR	a, b, c	00110bbb000001011BBBCCCCCAAAAAA
CMPYCHFR	a, b, u6	00110bbb010001011BBBuuuuuuAAAAAA
CMPYCHFR	b, b, s12	00110bbb100001011BBBssssssSSSSSS
CMPYCHFR<.cc>	b, b, c	00110bbb110001011BBBCCCCC0QQQQQ
CMPYCHFR<.cc>	b, b, u6	00110bbb110001011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYCHNFR

Function

Multiply two signed fractional complex conjugate operands. The result is rounded and saturated and returned without a fractional shift.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16((b.i*c.r - b.r*c.i));
a.r = SAT16(( b.i*c.i + b.i*c.i));
```

Instruction Format

op a,b,c

Syntax Example

CMPYCHNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply two signed fractional complex conjugate operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator. Return the saturated and rounded complex vector without a fractional shift in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
= 0;
addr0 = b.h0 * c.h0;
addr1 = b.h1 * c.h1;
addi0 = b.h1 * c.h0;
addi1 = b.h0 * -c.h1;
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```

CMPYCHNFR r0,r1,r2    ;Signed complex conjugated multiply
                      ; two complex vectors r1 and r2 and
                      ; return result in r0 with saturation and rounding
                      ;and returned without a fractional shifting

```

Syntax and Encoding

		Instruction Code
CMPYCHNFR	a, b, c	00110bbb000000101BBBCCCCCAAAAAA
CMPYCHNFR	a, b, u6	00110bbb010000101BBBuuuuuuAAAAAA
CMPYCHNFR	b, b, s12	00110bbb100000101BBBssssssSSSSSS
CMPYCHNFR<.cc>	b, b, c	00110bbb110000101BBBCCCCC0QQQQQ
CMPYCHNFR<.cc>	b, b, u6	00110bbb110000101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYHFR

Function

Multiply two signed fractional complex operands. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16 (RND16 ((b.i*c.r + b.r*c.i)<<1));
a.r = SAT16 (RND16 ((b.r*c.r - b.i*c.i)<<1));
```

Instruction Format

op a,b,c

Syntax Example

CMPYHFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply two signed fractional complex operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        if (BYTE_ORDER == little_endian) {
            addr0 = b.h0 * c.h0 << 1;
            addr1 = b.h1 * -c.h1 << 1;
            addi0 = b.h1 * c.h0 << 1;
            addi1 = b.h0 * c.h1 << 1;
        } else {
            addr0 = b.h1 * c.h0 << 1;
            addr1 = b.h0 * c.h1 << 1;
            addi0 = b.h0 * -c.h0 << 1;
            addi1 = b.h1 * c.h1 << 1;
        }
        acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        s                 |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        s                 |= sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        if (BYTE_ORDER == little_endian) {
            s |= sat((b.h0 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h1 * -c.h1 << 1), 31, &addr1);
            s |= sat((b.h1 * c.h0 << 1), 31, &addi0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addi1);
            addr1 = -addr1;
        } else {
            s |= sat((b.h1 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addr1);
            s |= sat((b.h0 * -c.h0 << 1), 31, &addi0);
            s |= sat((b.h1 * c.h1 << 1), 31, &addi1);
            addi0 = -addi0;
        }
}

```

```

    acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    if (BYTE_ORDER == little_endian) {
        addr0 = b.h0 * c.h0;
        addr1 = b.h1 * -c.h1;
        addi0 = b.h1 * c.h0;
        addi1 = b.h0 * c.h1;
    } else {
        addr0 = b.h1 * c.h0;
        addr1 = b.h0 * c.h1;
        addi0 = b.h0 * -c.h0;
        addi1 = b.h1 * c.h1;
    }
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);

```

```

else
    // guard bits are disabled
    acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;0
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo<<1,16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi<<1, 16), 31, &ahi);
    DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V ;
    a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```

CMPYHFR r0,r1,r2      ;Signed complex multiply
                      ; two complex vectors r1 and r2 and
                      ; return result in r0 with saturation and rounding

```

Syntax and Encoding

Instruction Code		
CMPYHFR	a, b, c	00110bbb00000011BBBCCCCCAAAAAA
CMPYHFR	a, b, u6	00110bbb010000011BBBuuuuuuAAAAAA
CMPYHFR	b, b, s12	00110bbb100000011BBBssssssSSSSSS
CMPYHFR<.cc>	b, b, c	00110bbb110000011BBBCCCCC0QQQQQ
CMPYHFR<.cc>	b, b, u6	00110bbb110000011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYHFMR

Function

Scale a signed fractional complex operand. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16 (b.i*c.i) <<1 ;
a.r = SAT16 ((b.r*c.i) <<1) ;
```

Instruction Format

op a,b,c

Syntax Example

CMPYHFMR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Scale a signed fractional complex operand, *b*, with the scalar fractional value specified in the *c* operand. Store the real result in the low accumulator and store the imaginary result in the high accumulator. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addr = b.h0 * c.h1 << 1;
        addi = b.h1 * c.h1 << 1;
        acclo = acc_add(0, addr, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        s                |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        s                |= sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h1 << 1), 31, &addr);
        s |= sat((b.h1 * c.h1 << 1), 31, &addi);
        acclo = acc_add(0, addr, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        s                |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        s                |= sat(round(acchi, 16), 31, &ahi);

```

CMPYHFMR

```

else
    // post-accumulate fractional shift mode
    addr = b.h0 * c.h1;
    addi = b.h1 * c.h1;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acclo = acc_add(0, addr, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s                |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(0, addi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s                |= sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acclo = acc_add(0, addr, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s                |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(0, addi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s                |= sat(round(acchi<<1, 16), 31, &ahi);
    DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V;
    a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```

CMPYHFMR r0,r1,r2    ;Signed complex scaling of r1 with scalar r2
                     ; return result in r0 with saturation and rounding

```

Syntax and Encoding

Instruction Code

CMPYHFMR	a,b,c	00110 bbb 00 011011 0 BBB CCCCC AAAAAA
CMPYHFMR	a,b,u6	00110 bbb 01 011011 0 BBB uuuuuu AAAAAA
CMPYHFMR	b,b,s12	00110 bbb 10 011011 0 BBB ssssss SSSSSS

CMPLYHFM R<.cc>	b, b, c	00110bbb110110110BBBCCCCC000000
CMPLYHFM R<.cc>	b, b, u6	00110bbb110110110BBBuuuuuu100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYHNFR

Function

Multiply two signed fractional complex operands. The result is rounded and saturated, and returned without a fractional shift.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16((b.i*c.r + b.r*c.i));
a.r = SAT16((b.r*c.r - b.i*c.i));
```

Instruction Format

op a,b,c

Syntax Example

CMPYHNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply two signed fractional complex operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in a little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in a big-endian mode. Return the saturated and rounded complex vector without fractional shift in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0;
    addr1 = b.h1 * -c.h1;
    addi0 = b.h1 * c.h0;
    addi1 = b.h0 * c.h1;
} else {
    addr0 = b.h1 * c.h0;
    addr1 = b.h0 * c.h1;
    addi0 = b.h0 * -c.h0;
    addi1 = b.h1 * c.h1;
}
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT = s | ACC0_GHI.V | ACC0_GLO.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMPLYHNFR r0,r1,r2    ;Signed complex multiply
                      ; two complex vectors r1 and r2 and
                      ; return result in r0 with saturation and rounding
                      ;and returned without a fractional shift
```

Syntax and Encoding

		Instruction Code
CMPLYHNFR	a, b, c	00110bbb00000001BBBCCCCCAAAAAA
CMPLYHNFR	a, b, u6	00110bbb010000001BBBuuuuuuAAAAAA
CMPLYHNFR	b, b, s12	00110bbb100000001BBBssssssSSSSSS
CMPLYHNFR<.cc>	b, b, c	00110bbb110000001BBBCCCCC0QQQQQ
CMPLYHNFR<.cc>	b, b, u6	00110bbb110000001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DIV

Function

2's Complement Integer Divide. For the non-DSP version, see [DIV](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

```
if (cc == true) {
  if ((c != 0) && ((b != 0x80000000) || c != 0xffffffff)) {
    r = b % c; // same sign as divisor
    q = b / c; // truncate to zero
    if (DSP_DIVSQRT == radix2) {
      ACC0_LO = r;
      ACC0_HI = q;
    }
    a = q;
    if (F == 1) {
      Z_flag = (a == 0) ? 1 : 0;
      N_flag = a < 0;
      V_flag = 0;
    }
  } else {
    if ((c == 0) && (STATUS32.DZ == 1)) {
      RaiseException (EV_DivZero);
    } else if (F == 1) {
      Z_flag = 0;
      N_flag = 0;
      V_flag = 1;
    }
  }
}
```

Instruction Format

op a, b, c

Syntax Example

DIV <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged

V • = Set if divisor is zero or if an arithmetic overflow occurs

Description

Signed integer division of the *b* operand by the *c* operand. The quotient is written to the high accumulator; the remainder is written to the low accumulator. The quotient is also returned in the destination register.

If the `STATUS32.DZ` bit is set to 1 when the divisor is zero, an `EV_DivZero` exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (*F*) bit for this instruction.

An arithmetic overflow happens if the smallest integer is divided by -1 (0x80000000/0xFFFFFFFF). The overflow flags also gets set if the divisor is 0.

When an overflow occurs or a division by 0 occurs, the accumulators and the destination register are not updated. When the divisor is non-zero, the sign of the remainder is the same as the sign of the *b* operand. The flags are updated only if the `.F` suffix is used.

This instruction is compatible to the ARCV2 `DIV` instruction with the additional side-effect of the accumulators getting updated, if DSP `DIVSQRT` radix-2 is configured.

Pseudo Code

```

if (cc == true) {
    if ((c != 0) && ((b != 0x80000000) || c != 0xffffffff)) {
        r = b % c; // same sign as divisor
        q = b / c; // truncate to zero
        if (DSP_DIVSQRT == radix2) {
            ACC0_LO = r;
            ACC0_HI = q;
        }
        a = q;
        if (F == 1) {
            Z_flag = (a == 0) ? 1 : 0;
            N_flag = a < 0;
            V_flag = 0;
        }
    } else {
        if ((c == 0) && (STATUS32.DZ == 1)) {
            RaiseException (EV_DivZero);
        } else if (F == 1) {
            Z_flag = 0;
            N_flag = 0;
            V_flag = 1;
        }
    }
}

```

dsp_divsq
 rt !=
 none
 or
 divrem !=
 none

Assembly Code Example

```
DIV r1,r2,r3 ; r1 is assigned r2 / r3
```

Syntax and Encoding

Instruction Code

DIV<.f>	a,b,c	00101bbb00000100FBBBCCCCC	AAAAAA
DIV<.f>	a,b,u6	00101bbb01000100FBBBuuuuuu	AAAAAA
DIV<.cc><.f>	b,b,c	00101bbb11000100FBBBCCCCC	0QQQQQ
DIV<.cc><.f>	b,b,u6	00101bbb11000100FBBBuuuuuu	1QQQQQ
DIV<.f>	b,b,s12	00101bbb10000100FBBBssssss	SSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DIVF

Function

Signed 32-bit fractional division.

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

```
if (cc) {
    acchi = (b << 31) / c;
    acclo = (b << 31) % c;
    a = acchi;
}.
```

Instruction Format

op a, b, c

Syntax Example

DIVF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if result is negative
C		= Unchanged
V	•	= Set if divisor is zero

Description

Signed fractional divide the 32-bits operands *b* and *c*, assuming $\text{abs}(b) \leq \text{abs}(c)$ or $B == -C$; the fractional quotient is stored in the high accumulator and the remainder is stored in the low accumulator. The quotient is also stored in the destination operand.

The `EV_DivZero` exception is raised if the divisor is zero and the `STATUS32.DZ` bit is set. An arithmetic overflow occurs if the absolute value of the numerator is greater than the absolute value of the denominator or if the denominator and numerator are equal, in which case the result is `0x80000000` or `0x7fffffff`. The overflow flag is set if the divisor is 0. For a division by 0, the accumulators and the destination register are not updated. If the divisor is non-zero, the sign of the remainder is the same as the sign of the *b* operand. The flags are only updated if the `.F` suffix is used.

Pseudo Code

```

if (c != 0) {                                     /* DIVF */
    if ((abs(b) < abs(c)) || (b == -c)) {
        ACC0_LO = (b<<31) % c; // same sign as divisor
        ACC0_HI = (b<<31) / c; // truncate to zero
    } else {
        ACC0_HI = (b < 0) == (c < 0) ? 0x7fffffff :
0x80000000;
        ACC0_LO = (b<<31) - ACC0_HI*c;
    }
    a = ACC0_HI;
    if (F == 1) {
        Z_flag = (a == 0) ? 1 : 0;
        N_flag = a < 0;
        V_flag = 0;
    }
    } else {
        if ((c == 0) && (STATUS32.DZ == 1)) {
            RaiseException (EV_DivZero);
        } else if (F == 1) {
            Z_flag = 0;
            N_flag = 0;
            V_flag = 1;
        }
    }
}
}

```

Assembly Code Example

```

DIVF r0,r1,r2                                ; Fractional divide r1 and r2 and
                                              ; return quotient in r0

```

Syntax and Encoding

Instruction Code

DIVF<.f>	a,b,c	00110 bbb 00 010000 F BBB CCCCC AAAAAA
DIVF<.f>	a,b,u6	00110 bbb 01 010000 F BBB uuuuuu AAAAAA
DIVF<.f>	b,b,s12	00110 bbb 10 010000 F BBB ssssss SSSSSS
DIVF<cc><.f>	b,b,u6	00110 bbb 11 010000 F BBB uuuuuu 1QQQQQ
DIVF<.cc><.f>	b,b,c	00110 110 11 010000 F 111 111110 0QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DIVU

Function

Unsigned integer division. For the non-DSP version, see [DIVU](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

```

if (cc == true)
{
    if (c != 0) {
        q = b / c;
        r = b % c;
        if (DSP_DIVSQRT == radix2) {
            // accumulator only updated for radix-2
            ACC0_LO = r;
            ACC0_HI = q;
        }
        a = q;
        if (F == 1) {
            Z_flag = (a == 0) ? 1 : 0;
            N_flag = 0;
            V_flag = 0;
        }
        else {
            if (STATUS32.DZ == 1) {
                RaiseException (EV_DivZero);
            }
            else if (F == 1) {
                Z_flag = 0;
                N_flag = 0;
                V_flag = 1;
            }
        }
    }
}

```

Instruction Format

op a, b, c

Syntax Example

DIVU <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always cleared
C		= Unchanged

V • = Set if divisor is zero

Description

Unsigned integer division of the *b* operand by the *c* operand. The quotient is written to the high accumulator and the remainder is written to the low accumulator. The quotient is also returned in the destination register.

If the `STATUS32.DZ` bit is set to 1 when the divisor is zero, an `EV_DivZero` exception is raised. In this case, the arithmetic flags are not modified regardless of the flag enable (F) bit for this instruction.

The overflow flags also gets set if the divisor is 0. When a division by 0 occurs, the accumulators and the destination register are not updated. When the divisor is non-zero, the sign of the remainder is the same as the sign of the *b* operand. The flags are updated only if the `.F` suffix is used.

This instruction is compatible to the ARCV2 DIV instruction with the additional side-effect of the accumulators getting updated, if DSP DIVSQRT radix-2 is configured.

Pseudo Code

```

if (cc == true) {
    if (c != 0) {
        q = b / c;
        r = b % c;
        if (DSP_DIVSQRT == radix2) {
            // accumulator only updated for radix-2
            ACC0_LO = r;
            ACC0_HI = q;
        }
        a = q;
        if (F == 1) {
            Z_flag = (a == 0) ? 1 : 0;
            N_flag = 0;
            V_flag = 0;
        }
    } else {
        if (STATUS32.DZ == 1) {
            RaiseException (EV_DivZero);
        } else if (F == 1) {
            Z_flag = 0;
            N_flag = 0;
            V_flag = 1;
        }
    }
}

```

dsp_divsqrt != none
or
divrem != none

Assembly Code Example

```
DIVU r1,r2,r3           ; r1 is assigned r2 / r3
```

Syntax and Encoding

Instruction Code		
DIVU<.f>	a, b, c	00101bbb00000101FBBBCCCCCAAAAAA
DIVU<.f>	a, b, u6	00101bbb01000101FBBBuuuuuuAAAAAA
DIVU<.cc><.f>	b, b, c	00101bbb11000101FBBBCCCCC0QQQQQ
DIVU<.cc><.f>	b, b, u6	00101bbb11000101FBBBuuuuuu1QQQQQ
DIVU<.f>	b, b, s12	00101bbb10000101FBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACH

Function

Dual 16x16 signed integer multiply and accumulate.

Extension Group

MPY_OPTION == 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
    a = result.w0;
    acc = result;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. w0 represents the lower 32-bits of an operand.

Instruction Format

op a, b, c

Syntax Example

DMACH <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	= Set if an overflow occurs during accumulation. This instruction never clears this flag.

Description

Multiply the lower 16 bits of the first and second operands, and multiply the higher 16 bits of the first and second operands to generate two 32-bit products. The two products are added to the wide accumulator to form the result. The least-significant 32 bits of the wide accumulator are assigned to the destination register. Flags are updated only if the set flags suffix (.F) is used.

Pseudo Code

```

accwide = accwide_org + b.h0 * c.h0 + b.h1 * c.h1;          /* DMACH */
if (DSP_CTRL.GE)      // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    V_flag |= rem != 0 && rem != -1;
    N_flag = accwide < 0;
}

```

Assembly Code Example

```
DMACH r1,r2,r3          ; dual 16x16 MAC of r2 and r3
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

Instruction Code

DMACH<.f>	a,b,c	00101bbb00010010FBBBCCCCC	AAAAAA
DMACH<.f>	a,b,u6	00101bbb01010010FBBBuuuuuu	AAAAAA
DMACH<.f>	b,b,s12	00101bbb10010010FBBBssssss	SSSSSS
DMACH<.cc><.f>	b,b,c	00101bbb11010010FBBBCCCCC	0QQQQQ
DMACH<.cc><.f>	b,b,u6	00101bbb11010010FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHBL

Function

Dual 16x8 signed integer multiply-accumulate operation of 16-bit vector elements of the first operand and the sign-extended first and second bytes of the second operand.

Extension Group

DSP Dual 16x8 MAC

Operation

$$a.h1 = accwide + b.h1 * sext(c.b1) + b.h0 * sext(c.b0);$$

Instruction Format

op a, b, c

Syntax Example

DMACHFBL <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	=Set if the accumulator overflows; this instruction never clears this flag

Description

Perform a signed integer multiplication of the higher 16 bits of the *b* operand and the sign-extended second byte of the *c* operand. Perform a signed integer multiplication the lower 16 bits of the *b* operand with the sign-extended first byte of the *c* operand. Add the two products to the wide accumulator and store the result in the wide accumulator. Return the least-significant 32 bits of the accumulator as the result. Flags are only updated if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

accwide += b.h0 * c.b0 + b.h1 * c.b1;                                /* DMACHBL
if (DSP_CTRL.GE) // only if guard bits are enabled                  */
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    V_flag |= rem != 0 && rem != -1;
    N_flag = accwide < 0;
}

```

Assembly Code Example

```

DMACHBL r0,r2,r3           ; Signed integer multiply-accumulate two
                           ;vectors r2 and r3 add the products ;into the
                           ;wide accumulator and return ;result in r0

```

Syntax and Encoding

Instruction Code

DMACHBL<.f>	a, b, c	00110 bbb 00 011000 FBBBCCCCC AAAAAA
DMACHBL<.f>	a, b, u6	00110 bbb 01 011000 FBBBuuuuuu AAAAAA
DMACHBL<.f>	b, b, s12	00110 bbb 10 011000 FBBBssssss SSSSSS
DMACHBL<.cc><.f>	b, b, c	00110 bbb 11 011000 FBBBCCCCC 0QQQQQ
DMACHBL<.cc><.f>	b, b, u6	00110 bbb 11 011000 FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHBM

Function

Dual 16x8 signed integer multiply-accumulate operation of 16-bit vector elements of the first operand and the sign-extended third and fourth bytes of the second operand.

Extension Group

DSP Dual 16x8 MAC

Operation

$$a.h1 = accwide + b.h1 * sext(c.b3) + b.h0 * sext(c.b2);$$

Instruction Format

op a, b, c

Syntax Example

DMACHBM <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	=Set if the accumulator overflows; this instruction never clears this flag

Description

Signed integer multiplication of the higher 16 bits of the *b* operand and the sign-extended fourth byte of the *c* operand; signed integer multiplication the lower 16 bits of the *b* operand with the sign-extended third byte of the *c* operand. Add the two products to the wide accumulator and store the result in the wide accumulator. Return the least-significant 32 bits of the accumulator as the result. Flags are only updated if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

accwide += b.h0 * c.b2 + b.h1 * c.b3;                                /* DMACHBM
if (DSP_CTRL.GE) // only if guard bits are enabled                  */
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    V_flag |= rem != 0 && rem != -1;
    N_flag = accwide < 0;
}

```

Assembly Code Example

```

DMACHBM r0,r2,r3           ; Signed integer multiply-accumulate two
                           ;vectors r2 and r3 add the products ;into the
                           ;wide accumulator and return ;result in r0

```

Syntax and Encoding

Instruction Code

DMACHBM<.f>	a, b, c	00110bbb00011001FBBBCCCCCAAAAA
DMACHBM<.f>	a, b, u6	00110bbb01011001FBBBuuuuuuAAAAA
DMACHBM<.f>	b, b, s12	00110bbb10011001FBBBssssssSSSSSS
DMACHBM<.cc><.f>	b, b, c	00110bbb11011001FBBBCCCCC000000
DMACHBM<.cc><.f>	b, b, u6	00110bbb11011001FBBBuuuuuu100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHF

Function

Dual 16x16 signed fractional multiplication and accumulation. The result is saturated.

Extension Group

DSP dual 16x16 MAC

Operation

$$a.h1 = \text{SAT16}((acchi += (b.h1 * c.h1 + b.h0 * c.h0) << 1));$$


Note

$h0$ (lower) and $h1$ (upper) represent the 16-bit elements of a 32-bit operand. $w0$ represents the lower 32-bits of an operand.

Instruction Format

op a, b, c

Syntax Example

DMACHF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result is too big to be stored in the destination

Description

Multiply the lower 16 bits of the first and second operands, and multiply the higher 16 bits of the first and second operands to generate two 32-bit products. The two products are added to the high accumulator to form the result. Return the 16-bit saturated fractional result to the destination register. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* DMACHF
s = 0;                                                                           */
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(acchi, addlo+addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 : ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(acchi, addhi+addlo, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi)
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;
        if (DSP_CTRL.GE)
            // guard bits are enabled
            acchi = acc_add(acchi, addhi+addlo, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
            ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
            ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 && ACC0_GHI.GUARD !=
0;
            ACC0_HI = acchi & 0xffff_ffff;
            v = sat(acchi<<1, 31, &ahi);
        else
            // guard bits are disabled
            acchi = acc_add(acchi, addhi+addlo, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
            ACC0_GHI.G = 0;
            ACC0_HI = acchi & 0xffff_ffff;
            v = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_0000);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}

```

Assembly Code Example

```

DMACHF r0,r1,r2                                ;signed fractional multiply-
                                                ;accumulate of two 16x16 vectors.
                                                ;The saturated result is stored
                                                ;in r0

```

Syntax and Encoding

		Instruction Code
DMACHF<.f>	a, b, c	00101bbb00101100FBBBCCCCCAAAAA
DMACHF<.f>	a, b, u6	00101bbb01101100FBBBuuuuuuAAAAA
DMACHF<.f>	b, b, s12	00101bbb10101100FBBBssssssSSSSSS
DMACHF<.cc><.f>	b, b, c	00101bbb11101100FBBBCCCCC0QQQQQ
DMACHF<.cc><.f>	b, b, u6	00101bbb11101100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHFR

Function

Dual 16x16 signed fractional multiplication and accumulation. The result is rounded and saturated.

Extension Group

DSP dual 16x16 MAC

Operation

$$a.h1 = \text{SAT16}(\text{RND16}((acchi += (b.h1 * c.h1 + b.h0 * c.h0) << 1)));$$


Note

$h0$ (lower) and $h1$ (upper) represent the 16-bit elements of a 32-bit operand. $w0$ represents the lower 32-bits of an operand.

Instruction Format

op a, b, c

Syntax Example

DMACHFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if the result is too big to be stored in the destination

Description

Multiply the lower 16 bits of the first and second operands, and multiply the higher 16 bits of the first and second operands. Add the two products to the high accumulator to form the result. Return the 16-bit saturated and rounded fractional result to the destination register. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(acchi, addlo+addhi, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(acchi, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    addlo = b.h0 * c.h0;
    addhi = b.h1 * c.h1;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acchi = acc_add(acchi, addhi+addlo, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acchi = acc_add(acchi, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_0000);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}

```

Assembly Code Example

```
DMACHFR r0,r1,r2           ;signed fractional multiply-
                           ;accumulate of two 16x16 vectors.
                           ;The saturated and rounded result
                           ;is stored ;in r0
```

Syntax and Encoding

Instruction Code

DMACHFR<.f>	a,b,c	00101bbb00101101FBBBCCCCC	AAAAAA
DMACHFR<.f>	a,b,u6	00101bbb01101101FBBBuuuuuu	AAAAAA
DMACHFR<.f>	b,b,s12	00101bbb10101101FBBBssssss	SSSSSS
DMACHFR<.cc><.f>	b,b,c	00101bbb11101101FBBBCCCCC	0QQQQQ
DMACHFR<.cc><.f>	b,b,u6	00101bbb11101101FBBBuuuuuu	100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHU

Function

Dual unsigned integer 16x16 multiply and accumulate.

Extension Group

MPY_OPTION == 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
    a = result.w0;
    acc = result;
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. `w0` represents the lower 32-bits of an operand.

Instruction Format

op a, b, c

Syntax Example

DMACHU <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Set if an overflow occurs during accumulation. This instruction never clears this flag.

Description

Multiply the lower 16 bits of the first and second operands, and multiply the higher 16 bits of the first and second operands. The two products are added to the wide accumulator to form the result. The least-significant 32 bits of the wide accumulator are assigned to the destination register. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
accwide = accwide_org + b.h0 * c.h0 + b.h1 * c.h1;           /* DMACHU */
if (DSP_CTRL.GE)      // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 72
else
    rem = accwide >> 64
if F {
    V_flag |= rem != 0
}
```

Assembly Code Example

```
DMACHU r1,r2,r3                ; dual 16x16 unsigned MAC of r2
                                ;and r3
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
DMACHU<.f>	a,b,c	00101bbb00010011FBBBCCCCCAAAAAA
DMACHU<.f>	a,b,u6	00101bbb01010011FBBBuuuuuuAAAAAA
DMACHU<.f>	b,b,s12	00101bbb10010011FBBBssssssSSSSSS
DMACHU<.cc><.f>	b,b,c	00101bbb11010011FBBBCCCCC0QQQQQ
DMACHU<.cc><.f>	b,b,u6	00101bbb11010011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYH

Function

Sum of dual 16x16 multiplication.

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. `w0` (lower) represents the 32-bit element of an operand.

Instruction Format

op a, b, c

Syntax Example

DMPYH <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set if accumulator is negative
C		= Unchanged
V	•	= Cleared

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the wide accumulator. Assign the least-significant 32 bits of the wide accumulator to the destination register.

Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
accwide = b.h0 * c.h0 + b.h1 * c.h1;                                /*DMPYH*/
if (DSP_CTRL.GE) // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag = 0;
    N_flag = accwide < 0;
}
```

Assembly Code Example

```
DMPYH r1,r2,r3                ; Sum of dual 16x16 multiply of r2
                               ;and r3
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

DMPYH<.f>	a,b,c	00101bbb00010000FBBBCCCCCAAAAAA
DMPYH<.f>	a,b,u6	00101bbb01010000FBBBuuuuuuAAAAAA
DMPYH<.f>	b,b,s12	00101bbb10010000FBBBssssssSSSSSS
DMPYH<.cc><.f>	b,b,c	00101bbb11010000FBBBCCCCC0QQQQQ
DMPYH<.cc><.f>	b,b,u6	00101bbb11010000FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHBL

Function

Dual 16x8 signed integer multiplication of 16-bit vector elements of the first operand and the sign-extended first and second bytes of the second operand.

Extension Group

DSP Dual 16x8 MAC

Operation

$$a.h1 = b.h1 * sext(c.b1) + b.h0 * sext(c.b0);$$

Instruction Format

op a, b, c

Syntax Example

DMPYHBL <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	=Set if the accumulator is too big to be represented in the result

Description

Perform a signed integer multiplication of the lower 16 bits of the *b* operand and the sign-extended first byte of the *c* operand. Perform a signed integer multiplication of the higher 16 bits of the *b* operand with the sign-extended second byte of the *c* operand. Add the two products and store the result in the wide accumulator. Return the least-significant 32 bits of the accumulator as the result. Flags are only updated if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
accwide = b.h0 * c.b0 + b.h1 * c.b1;                                /* DMPYHBL
if (DSP_CTRL.GE) // only if guard bits are enabled                */
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag = ((accwide & ~0x7fff_ffff) != 0) && ((accwide |
0x7fff_ffff) != -1);
    N_flag = accwide < 0;
}
```

Assembly Code Example

```
DMPYHBL r0,r2,r3           ; Signed integer multiplication of two vectors
                           ;r2 and r3, and add the products return result
                           ;in r0
```

Syntax and Encoding

Instruction Code		
DMPYHBL<.f>	a, b, c	00110bbb00010110FBBBCCCCC AAAAAA
DMPYHBL<.f>	a, b, u6	00110bbb01010110FBBBuuuuuu AAAAAA
DMPYHBL<.f>	b, b, s12	00110bbb10010110FBBBssssss SSSSSS
DMPYHBL<.cc><.f>	b, b, c	00110bbb11010110FBBBCCCCC 0QQQQQ
DMPYHBL<.cc><.f>	b, b, u6	00110bbb11010110FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHBM

Function

Dual 16x8 signed integer multiplication of 16-bit vector elements of the first operand and the sign-extended third and fourth bytes of the second operand.

Extension Group

DSP Dual 16x8 MAC

Operation

$$a.h1 = b.h1 * sext(c.b3) + b.h0 * sext(c.b2);$$

Instruction Format

op a, b, c

Syntax Example

DMPYHBM <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	=Set if the accumulator is too big to be represented in the result

Description

Perform a signed integer multiplication of the lower 16 bits of the *b* operand and the sign-extended third byte of the *c* operand. Signed integer multiplication of the higher 16 bits of the *b* operand with the sign-extended fourth byte of the *c* operand. Add the two products and store the result in the wide accumulator. Return the least-significant 32 bits of the accumulator as the result. Flags are only updated if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
accwide = b.h0 * c.b2 + b.h1 * c.b3;                                /* DMPYHBM
if (DSP_CTRL.GE) // only if guard bits are enabled                */
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag = ((accwide & ~0x7fff_ffff) != 0) && ((accwide |
    0x7fff_ffff) != -1);
    N_flag = accwide < 0;
}
```

Assembly Code Example

```
DMPYHBM r0,r2,r3          ; Signed integer multiplication of two vectors
                           ;r2 and r3, and add the products return result
                           ;in r0
```

Syntax and Encoding

Instruction Code		
DMPYHBM<.f>	a, b, c	00110bbb00010111FBBBCCCCC AAAAAA
DMPYHBM<.f>	a, b, u6	00110bbb01010111FBBBuuuuuu AAAAAA
DMPYHBM<.f>	b, b, s12	00110bbb10010111FBBBssssss SSSSSS
DMPYHBM<.cc><.f>	b, b, c	00110bbb11010111FBBBCCCCC 0QQQQQ
DMPYHBM<.cc><.f>	b, b, u6	00110bbb11010111FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHF

Function

Sum of dual 16x16 signed fractional multiplication. The result is saturated and stored.

Extension Group

DSP dual 16x16 MAC

Operation

```
acchi = b.h1*c.h1+b.h0*c.h0;
a.h1 = SAT16(acchi<<1);
```



Note

h_0 (lower) and h_1 (upper) represent the 16-bit elements of a 32-bit operand. w_0 (lower) represents the 32-bit element of an operand.

Instruction Format

op a, b, c

Syntax Example

DMPYHF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if result is negative
C		= Unchanged
V	•	= Set if the accumulator value is too big to be stored in the result

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the high accumulator. Return the saturated sum of products as a 16-bit fraction. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* DMPYHF */
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(0, addlo+addhi, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
else
    // post-accumulate fractional shift mode
    addlo = b.h0 * c.h0;
    addhi = b.h1 * c.h1;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acchi = acc_add(0, addhi+addlo, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi<<1, 31, &ahi);
    else
        // guard bits are disabled
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_0000);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}
```

Assembly Code Example

```
DMPYHF r0, r1, r2          ; Signed fractional
                             ; multiplication of vectors r2 and
                             ; r1. Store the saturated result
                             ; in r0
```

Syntax and Encoding

Instruction Code

DMPYHF<.f>	a,b,c	00101 bbb 00 101010 F BBB CCCCC AAAAAA
DMPYHF<.f>	a,b,u6	00101 bbb 01 101010 F BBB uuuuuu AAAAAA
DMPYHF<.f>	b,b,s12	00101 bbb 10 101010 F BBB ssssss SSSSSS
DMPYHF<.cc><.f>	b,b,c	00101 bbb 11 101010 F BBB CCCCC 0QQQQQ
DMPYHF<.cc><.f>	b,b,u6	00101 bbb 11 101010 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHFR

Function

Sum of dual 16x16 signed fractional multiplication. The result is rounded and saturated.

Extension Group

DSP dual 16x16 MAC

Operation

```
acchi = b.h1*c.h1+b.h0*c.h0;
a.h1 = SAT16(RND16((acchi<<1)));
```

Instruction Format

op a, b, c

Syntax Example

DMPYHFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if result is negative
C		= Unchanged
V	•	= Set if the accumulator value is too big to be stored in the result

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the high accumulator. Return the rounded and saturated sum of products as a 16-bit fraction. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(0, addlo+addhi, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    addlo = b.h0 * c.h0;
    addhi = b.h1 * c.h1;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acchi = acc_add(0, addhi+addlo, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_0000);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}

```

Assembly Code Example

```
DMPYHFR r0, r1, r2      ; Signed fractional ;multiplication of
                        ;vectors r2 and r1. Store the saturated and
                        ;rounded result in r0
```

Syntax and Encoding

Instruction Code

DMPYHFR<.f>	a,b,c	00101 bbb 00101011 FBBB CCCCC AAAAAA
DMPYHFR<.f>	a,b,u6	00101 bbb 01101011 FBBB uuuuuu AAAAAA
DMPYHFR<.f>	b,b,s12	00101 bbb 10101011 FBBB ssssss SSSSSS
DMPYHFR<.cc><.f>	b,b,c	00101 bbb 11101011 FBBB CCCCC 0QQQQQ
DMPYHFR<.cc><.f>	b,b,u6	00101 bbb 11101011 FBBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHU

Function

Sum of dual unsigned 16x16 multiplication.

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}
```



Note

$h0$ (lower) and $h1$ (upper) represent the 16-bit elements of a 32-bit operand. $w0$ (lower) represents the 32-bit element of an operand.

Instruction Format

op a, b, c

Syntax Example

DMPYHU <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Cleared

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the wide accumulator. Assign the least-significant 32 bits to the accumulator to the destination operand.

Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

accwide = b.h0 * c.h0 + b.h1 * c.h1;                                /* DMPYHU */
if (DSP_CTRL.GE) // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag = 0
}

```

Assembly Code Example

```
DMPYHU r1,r2,r3    ; sum of dual 16x16 unsigned multiply of r2 and r3
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

DMPYHU<.f>	a,b,c	00101bbb00010001FBBBCCCCCAAAAAA
DMPYHU<.f>	a,b,u6	00101bbb01010001FBBBuuuuuuAAAAAA
DMPYHU<.f>	b,b,s12	00101bbb10010001FBBBssssssSSSSSS
DMPYHU<.cc><.f>	b,b,c	00101bbb11010001FBBBCCCCC0QQQQQ
DMPYHU<.cc><.f>	b,b,u6	00101bbb11010001FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHWF

Function

Sum of dual 16x16 signed fractional multiplication. The result is a saturated 32-bit fraction.

Extension Group

DSP dual 16x16 MAC

Operation

```
acchi = b.h1*c.h1+b.h0*c.h0;
a.h1 = SAT32(acchi<<1);
```



Note

$h0$ (lower) and $h1$ (upper) represent the 16-bit elements of a 32-bit operand. $w0$ (lower) represents the 32-bit element of an operand.

Instruction Format

op a, b, c

Syntax Example

DMPYHWF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if result is negative
C		= Unchanged
V	•	= Set if the accumulator value is too big to be stored in the result

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the high accumulator. Return the saturated sum of products as a 32-bit fraction. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* DMPYHWF
s = 0;                                                                    */
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(0, addlo+addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;
        if (DSP_CTRL.GE)
            // guard bits are enabled
            acchi = acc_add(0, addhi+addlo, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
            ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
            ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
            ACC0_HI = acchi & 0xffff_ffff;
            v = sat(acchi<<1, 31, &ahi);
        else
            // guard bits are disabled
            acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
            ACC0_GHI.G = 0;
            ACC0_HI = acchi & 0xffff_ffff;
            v = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_ffff);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}

```

Assembly Code Example

```
DMPYHWF r0, r1, r2      ; Signed fractional ;multiplication of vectors r2
                        ;and r1. Store the 32-bit saturated result in r0
```

Syntax and Encoding

Instruction Code		
DMPYHWF<.f>	a,b,c	00101bbb00101000FBBBCCCCCAAAAAA
DMPYHWF<.f>	a,b,u6	00101bbb01101000FBBBuuuuuuAAAAAA
DMPYHWF<.f>	b,b,s12	00101bbb10101000FBBBssssssSSSSSS
DMPYHWF<.cc><.f>	b,b,c	00101bbb11101000FBBBCCCCC0QQQQQ
DMPYHWF<.cc><.f>	b,b,u6	00101bbb11101000FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

FLAGACC

Function

Copy accumulator flags to the STATUS32 register.

Extension Group

DSP Accumulator

Operation

```
switch (c.b1) {
0,2,3: STATUS32.ZNCV = ACC0_GHI.ZNCV;
1: STATUS32.ZNCV = ACC0_GLO.ZNCV;
}
```

Instruction Format

op c

Syntax Example

FLAGACC c

STATUS32 Flags Affected

Z	•	= From accumulator
N	•	= From accumulator
C	•	= From accumulator
V	•	= From accumulator

Description

Copy flags from the accumulator guard register to the STATUS32 register. The `c` operand consists of one field: the accumulator select.

Using the loop-counter (LP_COUNT/r60) as a source operand to the FLAGACC instruction results in an illegal instruction exception.

Pseudo Code

```
switch (c.b1) {                                     /* FLAGACC */
  case 0, 2, 3:
    STATUS32.ZNCV = ACC0_GHI.ZNCV;
    break;
  case 1:
    STATUS32.ZNCV = ACC0_GLO.ZNCV;
    break;
}
```

Assembly Code Example

```
FLAGACC r2      ; Get accumulator flags information from the fields in r2
```

Syntax and Encoding

Instruction Code		
FLAGACC	c	00101100001011111000CCCCC111111
FLAGACC	u6	00101100011011111000uuuuuu111111

GETACC

Function

Get the accumulator value

Extension Group

DSP Accumulator

Operation

```
b = shift_round_sat(acc, c);
```

Instruction Format

op b,c

Syntax Example

GETACC b, c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Arithmetic left shift of the accumulator including the guard bits; optionally round and saturate the result. The result can be MSB or LSB aligned.

If, `dsp_accshift==full`, the shift amount is saturated to a range of $-72 \leq \text{shamt} \leq +72$; if `dsp_accshift==limited`, the shift amount is saturated to a range of $-8 \leq \text{shamt} \leq +8$.

Example 1: Cast the wide accumulator (`DSP_CTRL.PA = 1`) to Q8 while saturating and rounding:

```
GETACC %r0, 24|0<<8|1<<10|1<<11|2<<12|3<<14
```

Example 2: Cast the high and low accumulator (`DSP_CTRL.PA = 0`) to a 2-way Q15 vector after shifting left by 2 while saturating and truncating:

```
GETACC %r0, (16+3)|3<<8|1<<10|1<<11|1<<12|2<<14 ; note 16+3 because post-accumulate ;shift-mode and shift left 2
```

Example 3: Cast the wide accumulator to an integer.

```
GETACC %r0, 32|0<<8|0<<10|1<<11|0<<12|0<<14
```

Instruction Encoding

Operand c	Description
Bits [7:0]	Shift amount; an 8 bit signed left-shift amount for reading the accumulator
Bits [9:8]	Accumulator select. 0: selects the wide accumulator; 1: selects the low accumulator; 2: selects the high accumulator; 3: selects dual hi/lo accumulator output.
Bit [10]	Enables saturation
Bit [11]	1: use arithmetic mode
Bit [12]	1: enable rounding; rounding is controlled by the DSP_CTRL.RM bit 0: enable truncation
Bit [13]	LSB/MSB alignment 0: MSB alignment 1: LSB alignment
Bit [15:14]	Rounding bit 0: result is 32 bits (rounding is at bit -1); 1: result is 24 bits rounding occurs at bit 7; 2: result is 16 bits and rounding occurs at bit 15; 3: result is 8b and rounding happen at bit 23. After rounding the result always resides in the reminaing MSBs, LSBs is set to 0.
Bit [16]	The GETACC instruction is sensitive to the PA mode bit; output data is shifted by one additional bit

Pseudo Code

```

// this code uses accumulator representation of 256b precision
if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)
    shamt = c.b0 < -72 ? -72 : (c.b0 > 72 ? 72 : c.b0);
else
    shamt = c.b0 < -8 ? -8 : (c.b0 > 8 ? 8 : c.b0);
accsel = C & 0x0000_0300 >> 8; // accumulator select:
                                // 0: wide accumulator
                                // 1: low accumulator
                                // 2: high accumulator
                                // 32: dual accumulator (returns 2x16b vector)
sat = C & 0x0000_0400 >> 10; // saturation enable
signed = C & 0x0000_0800 >> 11; // signed operation
rnd_en = C & 0x0000_1000 >> 12; // round mode
lsb = C & 0x0000_2000 >> 13; // LSB/MSB alignment
rndbyte = C & 0x0000_c000 >> 14; // index of byte to round
pa = C & 0x0001_0000 >> 16; // PA bit sensitive, one bit extra shift
if (rnd_en == 1)
    rndmode = DSP_CTRL.RM;
else
    rndmode = 0;
if (DSP_CTRL.GE)
    accwide = ACC0_GHI.GUARD << 64 | unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO);
    acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
    acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
else
    accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
    acchi = ACC0_HI;
    acclo = ACC0_LO;
if (accsel == 0)
    hi = accwide;
else
    hi = acchi << 32;
lo = acclo << 32;
if (pa && !DSP_CTRL.PA)
    // one extra bit left shift for post-accumulate shift mode
    shamt++;
// sign extend on left, zero extend on right
hiext = signed ? sext(hi) << 72 : ext(hi) << 72;
loext = signed ? sext(lo) << 72 : ext(lo) << 72;
// shift
shamt = shamt < -72 ? -72 : (shamt > 72 ? 72 : shamt);
hishift = shamt >= 0 ? hiext << shamt : hiext >> -shamt;
loshift = shamt >= 0 ? loext << shamt : loext >> -shamt;
// round at dot -8, -16, -24, -32
hirnd = round(hishift, 72+32+8*rndbyte, rndmode)>>72;
lornd = round(loshift, 72+32+8*rndbyte, rndmode)>>72;

```

```

// saturate at fractional dot
if (sat) {
    sh = sat(hirnd, 63, &hisat); // signed or unsigned saturation
    sl = sat(lornd, 63, &losat);
} else {
    hisat = hirnd;
    losat = lornd;
}
// mask of the LSBs that got rounded
switch (rndbyte) {
    3: hisat &= 0xff000000_00000000; losat &= 0xff000000_00000000;
    2: hisat &= 0xffff0000_00000000; losat &= 0xffff0000_00000000;
    1: hisat &= 0xffffffff00_00000000; losat &= 0xffffffff00_00000000;
}
// compose result
switch (accsel) {
    case 3: b.h1 = hisat >> 48; b.h0 = losat >> 48; DSP_CTRL.SAT |= sat & (sh|sl);
break;
    case 2, 0: b = hisat >> 32; DSP_CTRL.SAT |= sat & sh; break;
    case 1: b = losat >> 32; DSP_CTRL.SAT |= sat & sl; break;
}
// do LSB alignment of result
if (lsb) {
    if (signed) {
        switch (rndbyte) {
            3: b = b >> 24;
            2: b = b >> 16;
            1: b = b >> 8;
        }
    } else {
        switch (rndbyte) {
            3: b = (unsigned) b >> 24;
            2: b = (unsigned) b >> 16;
            1: b = (unsigned) b >> 8;
        }
    }
}
}

```

Assembly Code Example

```
GETACC r0,r2      ; Get accumulator value based on fields in r2; result to r0
```

Syntax and Encoding

Instruction Code

GETACC	b,c	00101 bbb 00 101111 0 BBB CCCCC 011000
GETACC	b,limm	00101 bbb 00 101111 0 BBB 111110 011000
GETACC	0,c	00101 110 00 101111 0 111 CCCCC 011000
GETACC	0,limm	00101 110 00 101111 0 111 111110 011000

GETACC	b,u6	00101bbb011011110BBBuuuuuu011000
GETACC	0,u6	00101110011011110111uuuuuu011000

MAC

Function

Signed 32x32 multiplication and accumulation.

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    a = result.w0;
    acc = result;
}
```

Instruction Format

op a, b, c

Syntax Example

MAC <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	= Set if an overflow occurs during accumulation. This instruction never clears this flag.

Description

Multiply the 32 bits of the first and second source operands to get a 64-bit product. This 64-bit product is added to the wide accumulator to form the result. Return the least-significant 32 bits of the wide accumulator in the destination register.

Flags are updated only if the set-flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
    unsigned(ACC0_LO)
else
    accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
accwide = accwide_org + b * c;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = (accwide & 0xffff_ffff);
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    N_flag = rem & 1;
    V_flag |= rem != -1 && rem != 0;
}
```

```
if (DSP_CTRL.GE)                                     /* MAC*/
    rem = accwide >> 72;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
else
    rem = accwide >> 64
a = (accwide & 0xffff_ffff_ffff_ffff);
if F {
    V_flag |= rem != 0;
}
```

Assembly Code Example

```
MAC r1,r2,r3    ; 32x32 multiply and accumulate r2 and r3 and store result
                ;in r1
```

Syntax and Encoding

Instruction Code

MAC<.f>	a,b,c	00101 bbb 00 0011110 F BBB CCCCC AAAAAA
MAC<.f>	a,b,u6	00101 bbb 01 0011110 F BBB uuuuuu AAAAAA
MAC<.f>	b,b,s12	00101 bbb 10 0011110 F BBB ssssss SSSSSS

MAC<.cc><.f>	b, b, c	00101bbb11001110FBBBCCCCC0QQQQQ
MAC<.cc><.f>	b, b, u6	00101bbb11001110FBBBuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACD

Function

Signed 32x32 multiplication and accumulation. Returns a 64-bit result.

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    A = result;
    acc = result;
}
```

Instruction Format

op A, b, c

Syntax Example

MACD <.f> A,b,c

Timing Characteristics

Single cycle

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	= Set if an overflow occurs during accumulation. This instruction never clears this flag.

Description

Multiply the 32 bits of the first and second source operands to get a 64-bit product. This 64-bit product is added to the wide accumulator to form the result. Return the least-significant 64 bits of the wide accumulator in the destination register.

Flags are updated only if the set-flags suffix (.F) is used.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
    unsigned(ACC0_LO)
else
    accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
accwide = accwide_org + b * c;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = (accwide & 0xffff_ffff);
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    N_flag = rem & 1;
    V_flag |= rem != -1 && rem != 0;
}
```

Assembly Code Example

```
MACD r0,r2,r3      ; 32x32y64 multiply-accumulate r2 and r3 and store
                   ;result in (r1,r0)
```

Syntax and Encoding

Instruction Code		
MACD<.f>	a,b,c	00101bbb00011010FBBBCCCCC0AAAAAA
MACD<.f>	a,b,u6	00101bbb01011010FBBBuuuuuuAAAAAA
MACD<.f>	b,b,s12	00101bbb10011010FBBBssssssSSSSSS
MACD<.cc><.f>	b,b,c	00101bbb11011010FBBBCCCCC0QQQQQ0
MACD<.cc><.f>	b,b,u6	00101bbb11011010FBBBuuuuuu1QQQQQ0

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACDF

Function

32x32 signed fractional multiplication and accumulation.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide += b * c;
    A = SAT64(accwide<<1);
}
```



Note

A is a register pair representing a 64-bit operand.

Instruction Format

op A, b, c

Syntax Example

MACDF <.f> A,b,c

Timing Characteristics

Two cycle

STATUS32 Flags Affected

Z		• = Set if the result is zero
N		• = Set to the resulting sign is negative
C		= Unchanged
V		• = Set if the result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands. The product is added to the wide accumulator to form the result. The saturated least-significant 64-bits of the result is then assigned to the destination register pair.

If the result saturates, the sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix. Flags are updated only if the set-flags suffix (`.F`) is used.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MACDF*/
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) <<
32 | unsigned(ACC0_LO)
        ACC0_GHI.V      = sat_en(accwide + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD !=
-1 : ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide + add, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(accwide, 63, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) <<
32 | unsigned(ACC0_LO)
        ACC0_GHI.V      = sat_en(accwide + b * c, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD !=
-1 && ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide + b * c, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(accwide << 1, 63, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}
```

Assembly Code Example

```
MACDF r0,r2,r3    ; signed fractional 32x32 multiply-accumulate r2 and r3
                  ;and store the saturated result in (r1,r0)
```

Syntax and Encoding

Instruction Code		
MACDF<.f>	a,b,c	00110bbb00010011FBBBCCCCC0AAAAAA
MACDF<.f>	a,b,u6	00110bbb01010011FBBBuuuuuuAAAAAA
MACDF<.f>	b,b,s12	00110bbb10010011FBBBssssssSSSSSS
MACDF<.cc><.f>	b,b,c	00110bbb11010011FBBBCCCCC0QQQQQ
MACDF<.cc><.f>	b,b,u6	00110bbb11010011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACDU

Function

Unsigned 32x32 multiplication and accumulation. Return a 64-bit result.

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + unsigned(b * c);
    A = result;
    acc = result;
}
```

Instruction Format

op A, b, c

Syntax Example

MACDU <.f> A,b,c

Timing Characteristics

Two cycle

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V	•	= Set if the accumulator overflows. This instruction never clears this flag.

Description

Multiply the 32 bits of the first and second source operands to get a 64-bit product. This 64-bit product is added to the wide accumulator to form the result. Return the least-significant 64 bits of the wide accumulator in the destination register.

Flags are updated only if the set-flags suffix (.F) is used.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MACDU*/
if (DSP_CTRL.GE)
    accwide_org = unsigned(ACC0_GHI.GUARD<<64) |
    unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO)
else
    accwide_org = unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO)
accwide = accwide + b * c;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = (accwide & 0xffff_ffff_ffff_ffff);
if F {
    V_flag |= sign(accwide) == 0 && sign(accwide_org) == 1;
}
```

Assembly Code Example

```
MACDU r0,r2,r3    ; 32x32y64 unsigned multiply-accumulate r2 and r3 and
                  ;store the result is stored in (r1, r0)
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

MACDU<.f>	a,b,c	00101bbb00011011FBBBCCCCC	AAAAAA
MACDU<.f>	a,b,u6	00101bbb01011011FBBBuuuuuu	AAAAAA
MACDU<.f>	b,b,s12	00101bbb10011011FBBBssssss	SSSSSS
MACDU<.cc><.f>	b,b,c	00101bbb11011011FBBBCCCCC	000000
MACDU<.cc><.f>	b,b,u6	00101bbb11011011FBBBuuuuuu	100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,r0,r1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACF

Function

Signed 32x32 fractional multiplication and accumulation. Return a saturated 32-bit result.

Extension Group

DSP 32x32 MAC

Operation

```
if (cc) {
    accwide += b * c;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set to the resulting sign is negative
C		= Unchanged
V	•	= Set if the result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands. The product is added to the wide accumulator to form the result. The truncated and saturated second 32-bit word of the result is then assigned to the destination register.

The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. Flags are updated only if the set-flags suffix (`.F`) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* MACF*/
s= 0;
if (DSP_CTRL.PA)
  if (DSP_CTRL.GE)
    add = b * c << 1;
    accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
    ACC0_GHI.V = sat_en(accwide_org + add, 71, &accwide);
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
    ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
  else
    s = sat((b * c << 1), 63, &add);
    accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
    ACC0_GHI.V = sat(accwide_org + add, 63, &accwide);
    ACC0_GHI.G = 0;
    v = sat(accwide >> 32, 31, &a);
  else
    if (DSP_CTRL.GE)
      accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
      ACC0_GHI.V = sat_en(accwide_org + b * c, 71, &accwide);
      ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
      ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
      accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
      ACC0_GHI.V = sat(accwide_org + b * c, 63, &accwide);
      ACC0_GHI.G = 0;
      v = sat(accwide >> 31, 31, &a);
    DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
    ACC0_HI = (accwide >> 32) & 0xffff_ffff;
    ACC0_LO = accwide & 0xffff_ffff;
    ACC0_GHI.Z = accwide == 0;
    ACC0_GHI.N = accwide < 0;
    if F {
      Z_flag = a = 0;
      N_flag = a < 0;
      V_flag = v;
    }

```

Assembly Code Example

```

MACF r0,r1,r2      ; signed fractional 32x32 multiply-accumulate r1 and r2
                   ;and store the saturated result in r0

```

Syntax and Encoding

		Instruction Code
MACF<.f>	a, b, c	00110bbb00001100FBBBCCCCCAAAAA
MACF<.f>	a, b, u6	00110bbb01001100FBBBuuuuuuAAAAA
MACF<.f>	b, b, s12	00110bbb10001100FBBBssssssSSSSSS
MACF<.cc><.f>	b, b, c	00110bbb11001100FBBBCCCCC0QQQQQ
MACF<.cc><.f>	b, b, u6	00110bbb11001100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACFR

Function

Signed 32x32 fractional multiplication and accumulation. Return a saturated and rounded 32-bit result.

Extension Group

DSP 32x32 MAC

Operation

```
if (cc) {
    accwide += b * c;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set to the resulting sign is negative
C		= Unchanged
V	•	= Set if the result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands. The product is added to the wide accumulator to form the result. The rounded and saturated higher 32 bits of the result is then assigned to the destination register.

The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. Flags are updated only if the set-flags suffix (`.F`) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MACFR*/
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI)
        << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide_org + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD !=
-1 : ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide_org + add, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(round(accwide,32) >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI)
        << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide_org + b * c, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD !=
-1 : ACC0_GHI.GUARD != 0;
    else
        accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide_org + b * c, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(round(accwide,31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}
```

Assembly Code Example

```
MACFR r0,r1,r2    ; signed fractional 32x32 multiply-accumulate r1 and r2
                  ;and store the saturated result in r0
```

Syntax and Encoding

		Instruction Code
MACFR<.f>	a,b,c	00110bbb00001101FBBBCCCCC AAAAAA
MACFR<.f>	a,b,u6	00110bbb01001101FBBBuuuuuu AAAAAA
MACFR<.f>	b,b,s12	00110bbb10001101FBBBssssss SSSSSS
MACFR<.cc><.f>	b,b,c	00110bbb11001101FBBBCCCCC 0QQQQQ
MACFR<.cc><.f>	b,b,u6	00110bbb11001101FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACU

Function

Unsigned 32x32 multiplication and accumulation.

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    a = result.w0;
    acc = result;
}
```



Note

w0 represents the lower 32-bits of a 64-bit result.

Instruction Format

op a, b, c

Syntax Example

MACU <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Set if the accumulator overflows. This instruction never clears this flag.

Description

Perform an unsigned multiplication of the 32 bits of the first and second source operands to get a 64-bit product. This 64-bit product is added to the wide accumulator to form the result. Return the least-significant 32 bits of the wide accumulator in the destination register.

Flags are updated only if the set-flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MACU*/
if (DSP_CTRL.GE)
    accwide_org = unsigned(ACC0_GHI.GUARD) << 64 |
    unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO)
else
    accwide_org = unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO)
accwide = accwide_org + b * c;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 72
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
else
    rem = accwide >> 64
a = (accwide & 0xffff_ffff);
if F {
    V_flag |= rem != 0;
}

```

Assembly Code Example

```

MACU r1,r2,r3                                ; 32x32 unsigned multiplication
                                              ;and accumulation of r2 and r3
                                              ;and store result in r1

```

Syntax and Encoding

Instruction Code

MACU<.f>	a,b,c	00101bbb000001111FBBBCCCCC	AAAAAA
MACU<.f>	a,b,u6	00101bbb010001111FBBBuuuuuu	AAAAAA
MACU<.f>	b,b,s12	00101bbb100001111FBBBssssss	SSSSSS
MACU<.cc><.f>	b,b,c	00101bbb110001111FBBBCCCCC	0QQQQQ
MACU<.cc><.f>	b,b,u6	00101bbb110001111FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHFM

Function

Signed 32x16 fractional multiplication and accumulation.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h1 <<32;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHFM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the saturated most-significant 32 bits of the accumulator, without guard bits. The sticky `DSP_CTRL.SAT` flag is set if saturation occurs, regardless of the `.F` suffix. Other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            ACC0_GHI.V = sat_en(accwide + b * c.h1 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            ACC0_GHI.V = sat(accwide + b * c.h1 << 16, 63, &accwide); v =
sat(accwide>>31, 31, &a);
            ACC0_GHI.G = 0;
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
/* MACWHFM
*/
```

Assembly Code Example

```
MACWHFM r0,r2,r3      ; Signed fractional multiply-accumulate r2 and
                      ;r3 and return result in register r0
```

Syntax and Encoding

		Instruction Code
MACWHFM<.f>	a, b, c	00110bbb00100010FBBBCCCCC AAAAAA
MACWHFM<.f>	a, b, u6	00110bbb01100010FBBBuuuuuu AAAAAA
MACWHFM<.f>	b, b, s12	00110bbb10100010FBBBssssss SSSSSS
MACWHFM<.cc><.f>	b, b, c	00110bbb11100010FBBBCCCCC 0QQQQQ
MACWHFM<.cc><.f>	b, b, u6	00110bbb11100010FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHFMR

Function

Signed 32x16 fractional multiplication and accumulation. The result is rounded and saturated.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h1 <<32;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHFMR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the rounded and saturated most-significant 32 bits of the accumulator without guard bits. The sticky `DSP_CTRL.SAT` flag is set if saturation occurs, regardless of the `.F` suffix. Other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            ACC0_GHI.V = sat_en(accwide + b * c.h1 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            ACC0_GHI.V = sat(accwide + b * c.h1 << 16, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(round(accwide, 31) >> 31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
    }
    /* MACWHFMR
    */
```

Assembly Code Example

```
MACWHFMR r0,r2,r3      ; Signed fractional multiply-accumulate r2 and
                        ;r3 and return result in register r0
```


Syntax and Encoding

		Instruction Code
MACWHFMR<.f>	a, b, c	00110bbb00100011FBBBCCCCC AAAAAA
MACWHFMR<.f>	a, b, u6	00110bbb01100011FBBBuuuuuu AAAAAA
MACWHFMR<.f>	b, b, s12	00110bbb10100011FBBBssssss SSSSSS
MACWHFMR<.cc><.f>	b, b, c	00110bbb11100011FBBBCCCCC 000000
MACWHFMR<.cc><.f>	b, b, u6	00110bbb11100011FBBBuuuuuu 100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHL

Function

Signed 32x16 integer multiplication and accumulation.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h0;
    a = accwide;
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHL <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	=Set if overflow occurs during accumulation; this instruction never clears this flag

Description

Signed integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the least-significant 32 bits of the accumulator. Flags are updated only if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
    accwide += b * c.h0;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
else
    accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
    accwide += b * c.h0;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    N_flag = accwide < 0;
    V_flag |= rem != -1 && rem != 0;
}
```

Assembly Code Example

```
MACWHL r0,r2,r3      ; Signed integer multiply-accumulate r2 and r3
                     ;and return result in register r0
```

Syntax and Encoding

Instruction Code

MACWHL<.f>	a,b,c	00110bbb00011101FBBBCCCCC	AAAAAA
MACWHL<.f>	a,b,u6	00110bbb01011101FBBBuuuuuu	AAAAAA
MACWHL<.f>	b,b,s12	00110bbb10011101FBBBssssss	SSSSSS
MACWHL<.cc><.f>	b,b,c	00110bbb11011101FBBBCCCCC	0QQQQQ
MACWHL<.cc><.f>	b,b,u6	00110bbb11011101FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHFL

Function

Signed 32x16 fractional multiplication and accumulation.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h0 <<32;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHFL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* MACWHFL */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            ACC0_GHI.V = sat_en(accwide + b * c.h0 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            ACC0_GHI.V = sat(accwide + b * c.h0 << 16, 63, &accwide); v =
sat(accwide>>31, 31, &a);
            ACC0_GHI.G = 0;
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }

```

Assembly Code Example

```

MACWHFL r0,r2,r3          ; Signed fractional multiply-accumulate r2 and r3
                           ;and return result in register r0

```

Syntax and Encoding

		Instruction Code
MACWHFL<.f>	a, b, c	00110bbb00100110FBBBCCCCC(AAAAAA)
MACWHFL<.f>	a, b, u6	00110bbb01100110FBBBuuuuuu(AAAAAA)
MACWHFL<.f>	b, b, s12	00110bbb10100110FBBBssssss(SSSSSS)
MACWHFL<.cc><.f>	b, b, c	00110bbb11100110FBBBCCCCC(0QQQQQ)
MACWHFL<.cc><.f>	b, b, u6	00110bbb11100110FBBBuuuuuu(1QQQQQ)

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHFLR

Function

Signed 32x16 fractional multiplication and accumulation with rounding.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h0 << 32;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHFLR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the rounded and saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* MACWHFLR */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + b * c.h0 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + b * c.h0 << 16, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide, 31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MACWHFLR r0,r2,r3          ; Signed fractional multiply-accumulate r2 and r3 and
                           ; return the rounded and saturated result in register r0

```

Syntax and Encoding

		Instruction Code
MACWHFLR<.f>	a, b, c	00110bbb00100111FBBBCCCCC AAAAAA
MACWHFLR<.f>	a, b, u6	00110bbb01100111FBBBuuuuuu AAAAAA
MACWHFLR<.f>	b, b, s12	00110bbb10100111FBBBssssss SSSSSS
MACWHFLR<.cc><.f>	b, b, c	00110bbb11100111FBBBCCCCC 0QQQQQ
MACWHFLR<.cc><.f>	b, b, u6	00110bbb11100111FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHKL

Function

Signed 32x16 integer multiplication and accumulation and shift-right the lower bits

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    acchi += b * c.h0 >> 16;
    a = acchi;
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHKL <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	=Set if the accumulator overflows; this instruction never clears this flag

Description

Signed integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and add the shifted product to the high accumulator. Return the least-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MACWHKL */
if (DSP_CTRL.GE)
    acchi = sext(ACC0_GHI.GUARD)<<32 | unsigned(ACC0_HI)
    acchi += (b * c.h0) >> 16;
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
else
    acchi = ACC0_HI;
    acchi += (b * c.h0) >> 16;
ACC0_HI = acchi & 0xffff_ffff;
a = acchi & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = acchi >> 39;
else
    rem = acchi >> 31;
if F {
    N_flag = rem & 1;
    V_flag |= rem != -1 && rem != 0;
}

```

Assembly Code Example

```

MACWHKL r0,r2,r3                ; Signed integer multiply-accumulate r2 and r3
                                ; shift result by 16 bits, and return the shifted
                                ; result in register r0

```

Syntax and Encoding

Instruction Code

MACWHKL<.f>	a,b,c	00110bbb00101000FBBBCCCCC	AAAAAA
MACWHKL<.f>	a,b,u6	00110bbb01101000FBBBuuuuuu	AAAAAA
MACWHKL<.f>	b,b,s12	00110bbb10101000FBBBssssss	SSSSSS
MACWHKL<.cc><.f>	b,b,c	00110bbb11101000FBBBCCCCC	0QQQQQ
MACWHKL<.cc><.f>	b,b,u6	00110bbb11101000FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHKUL

Function

Unsigned 32x16 integer multiplication and accumulation and shift-right the lower bits

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    acchi += b * c.h0 >> 16;
    a = acchi;
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHKUL <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	=Set if the accumulator overflows; this instruction never clears this flag

Description

Unsigned integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and add the shifted product to the high accumulator. Return the least-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used.

Pseudo Code

```

if (DSP_CTRL.GE)                                     /* MACWHKUL */
    acchi = unsigned(ACC0_GHI.GUARD)<<32 | unsigned(ACC0_HI);
    acchi += (b * c.h0) >> 16;
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    s = acchi >> 40 != 0;
else
    acchi = unsigned(ACC0_HI);
    acchi += (b * c.h0) >> 16;
    s = acchi >> 32 != 0;
ACC0_HI = acchi & 0xffff_ffff;
a = acchi & 0xffff_ffff;
if F {
    V_flag |= s;
}

```

Assembly Code Example

```

MACWHKUL r0,r2,r3      ; Unsigned integer multiply-accumulate r2 and r3
                        ; shift result by 16 bits, and return the shifted
                        ; result in register r0

```

Syntax and Encoding

Instruction Code		
MACWHKUL<.f>	a, b, c	00110bbb00101001FBBBCCCCC AAAAAA
MACWHKUL<.f>	a, b, u6	00110bbb01101001FBBBuuuuuu AAAAAA
MACWHKUL<.f>	b, b, s12	00110bbb10101001FBBBssssss SSSSSS
MACWHKUL<.cc><.f>	b, b, c	00110bbb11101001FBBBCCCCC 0QQQQQ
MACWHKUL<.cc><.f>	b, b, u6	00110bbb11101001FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHUL

Function

Unsigned 32x16 integer multiplication and accumulation.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += (unsigned) (b * c.h0);
    a = accwide;
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHUL <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Unchanged
C		= Unchanged
V	•	=Set if the accumulator overflows.; sticky

Description

Unsigned integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the least-significant 32 bits of the accumulator. Flags are updated only if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide = unsigned(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
    accwide += b * c.h0;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
    s = accwide >> 72 != 0;
else
    accwide = unsigned(ACC0_HI << 32) | unsigned(ACC0_LO);
    accwide += b * c.h0;
    s = accwide >> 64 != 0;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag |= s;
}
/* MACWHUL
*/

```

Assembly Code Example

```

MACWHUL r0,r2,r3      ; Unsigned integer multiply-accumulate r2 and r3
                      ;and return result in register r0

```

Syntax and Encoding

Instruction Code

MACWHUL<.f>	a, b, c	00110bbb00011111FBBBCCCCC AAAAAA
MACWHUL<.f>	a, b, u6	00110bbb01011111FBBBuuuuuu AAAAAA
MACWHUL<.f>	b, b, s12	00110bbb10011111FBBBssssss SSSSSS
MACWHUL<.cc><.f>	b, b, c	00110bbb11011111FBBBCCCCC 0QQQQQ
MACWHUL<.cc><.f>	b, b, u6	00110bbb11011111FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MODIF

Function

Update an address pointer based on an operand modifier value

Extension Group

AGU

Operation

```
AGU_AUX_AP[c.ptr_reg] = next_addr(c);
```

Instruction Format

op c



Note

You cannot use the XY operands as a source or destination with this instruction.

Syntax Example

MODIF c

Timing Characteristics

Issue one instruction every two cycles

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Interpret the `c` operand value as a modifier and update the corresponding address pointer. This instruction does not return any value and does not update any flags.

This instruction does not raise any exception with an illegal modifier or an illegal address. However, this instruction implements the following checks:

- If an operand refers to a non-existing address pointer (`AGU_AUX_APx`), the associated address pointer is not updated.
- If an operand refers to a non-existing modifier (`AGU_AUX_MODx`), the operand is not updated.

- If an operand refers to a non-existing offset register (AGU_AUX_OS_x), the associated address pointer is not updated.
- If an operand refers to an illegal F_x ($\text{AGU\_AUX\_MOD}_x[10:6]$) value, the associated address pointer is not updated. Illegal F_x values are 15 and greater than 22.
- If a source operand uses $\text{AGU\_AUX\_MOD}_x[5:4] == 3$, the associated address pointer is not updated.
- If a source operand uses $2^{\text{VW}} < \text{container size}$, the associated address pointer is not updated. Where, VW is the $\text{AGU\_AUX\_MOD}_x[5:4] == 3$ field.

Pseudo Code

```

// pseudocode for updating the address pointer value                                /* MODIF*/
// modifier fields as per section 8.3.7
void update_addr(bit_vector<5> mod_reg) { // the modifier index
    mod_t mod; // modifier value (struct as per AUX reg definition)
    mod = AGU_AUX_MOD[mod_reg];
    apply_modifier(mod);
}
int32 bitrev(int32 a) {
    // reverse bit order in an address vector
    int32 r = 0;
    for (int i = 0; i < ADDR_WIDTH; i++) { // ADDR_WIDTH = unnumber of bits in
        address
            r = (r << 1) | (a & 1);
            a >>= 1;
    }
    return r;
}
uint32 log2up(uint32 a) {
    // find next power of 2 bigger than or equal to a
    uint32 r = 0;
    while ((1 << r) < a) r++;
    return r;
}
void apply_modifier(mod_t mod) { // the modifier
    uint32 addr; // current address
    uint32 addr_nxt; // next address
    int32 offset; // signed offset
    int32 wrap; // unsigned wrap boundary, minus 1
    int32 mask; // mask for wrapping
    // retrieve pointer value
    addr = AGU_AUX_AP[mod.ptr_reg];
    // get offset
    if ((mod.opc % 2) == 0) {
        offset = AGU_AUX_OS[mod.offset_reg];
    } else {
        offset = unsigned(mod.offset_imm);
    }
    // get wrap boundary or scale offset
    if (mod.opc == 4) {
        wrap = AGU_AUX_OS[mod.wrap_reg];
    } else if (mod.opc == 5 || mod.opc == 6) {
        wrap = unsigned(power(2, mod.wrap_imm+1))-1;
    } else {
        offset = offset << mod.sc;
    }
}

```

```

// flip pointer and offset bits if bit-reverse
if (mod.opc == 2 || mod.opc == 3) {
    addr    = bitrev(addr);
    offset  = bitrev(offset);
}
// optionally negate offset
if (mod.dir)
    offset = -offset;
addr_nxt = addr + offset;
if (mod.opc == 2 || mod.opc == 3) {
    // flip back pointer bits if bit-reverse, avoid carry prop into lsbs
    addr_nxt = bitrev(addr_nxt); // bitrev is flipping bits in 32b word
    lsbmask = -power(2, mod.vw-mod.cs+mod.ds); // create a mask to clear
lsbs
    addr_nxt &= lsbmask;
} else if (mod.opc >= 4) {
    // wrap
    mask = power(2, log2up(wrap+1))-1; // log2up is base 2 logarithm
rounded up
    if (((addr_nxt & ~mask) != (addr & ~mask)) ||
        ((addr_nxt & mask) > wrap)) {
        // went through the wrap boundary
        if (offset > 0) {
            addr_nxt -= wrap - 1; // went over max
        } else {
            addr_nxt += wrap + 1; // went under min
        }
    }
} // else transparent
// update address with next value
AGU_AUX_AP[mod.ptr_reg] = addr_nxt;
}

```

Assembly Code Example

```

MODIF c          ; Update the address pointer based on the immediate
                  ; modifier value

```

Syntax and Encoding

Instruction Code

MODIF	c	00101101001011110000CCCCC111111
MODIF	u6	00101101011011110000uuuuuu111111

MODAPP

Function

Apply a modifier without accessing the memory

Extension Group

AGU

Operation

```
if (c == agu_ux) {
    AGU_AUX_AP[c.ptr_reg] = next_addr(c);
}
if (b == agu_ux) {
    AGU_AUX_AP[b.ptr_reg] = next_addr(b);
}
```

Instruction Format

op b, c

Syntax Example

MODAPP b, c

MODAPP 0, c

Timing Characteristics

Issue one instruction every two cycles

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

This instruction takes two modifiers as input. If the operands are AGU window registers, that is, $32 \leq B < \text{AGU\_MOD\_NUM} + 32$, $32 \leq C < \text{AGU\_MOD\_NUM} + 32$, the associated modifier is applied to update the address pointer. If an operand is a non AGU window register or if the operand is an immediate, the address pointer is not updated. This instruction does not return a value or update the flags.

This instruction does not raise any exception with an illegal modifier or an illegal address. However, this instruction implements the following checks:

- If an operand refers to a non-existing modifier register (AGU_AUX_MODx), the associated address pointer is not updated.
- If an operand refers to a non-existing address pointer (AGU_AUX_APx), the associated address pointer is not updated.
- If an operand refers to a non-existing offset register (AGU_AUX_OSx), the associated address pointer is not updated.
- If an operand refers to an illegal Fx ($AGU\_AUX\_MODx[10:6]$) value, the associated address pointer is not updated. Illegal Fx values are 15 and greater than 22.
- If a source operand uses $AGU\_AUX\_MODx[5:4] == 3$, the associated address pointer is not updated.
- If a source operand uses $2^{VW} < \text{container size}$, the associated address pointer is not updated. Where, VW is the $AGU\_AUX\_MODx[5:4] == 3$ field.

Pseudo Code

```
// BX is the index of the B operand register
if (BX >= 32 && BX <= AGU_MOD_NUM+32) {
    m = AGU_AUX_MOD[BX-32];
    apply_modifier(m);
}
// CX is the index of the C operand register
if (CX >= 32 && CX <= AGU_MOD_NUM+32) {
    n = AGU_AUX_MOD[CX-32];
    apply_modifier(n);
}
```

/*
MODAPP*/

Assembly Code Example

```
MODAPP agu_u2, agu_u3      ;Apply the second and third modifier
                           ;updating the associated address pointer
```

Syntax and Encoding

Instruction Code

MODAPP	b, c	00101bbb001011110BBBCCCCC111110
MODAPP	b, u6	00101bbb011011110BBBuuuuuu111110

MPY

Function

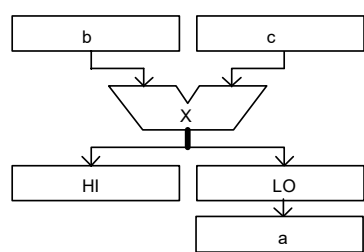
32 x 32 Signed Multiply, returning least-significant 32-bit word of the result. For the non-DSP version, see [MPY](#), [MPY_S](#).

Extension Group

MPY_OPTION > 1 or has_dsp==true

Operation

if (cc) a = (signed) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPY<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 64-bit result is set
C		= Unchanged
V	•	=Set when the result cannot be represented as a 32b signed integer

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers
t = b * c;
a = (t & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = t < 0;
    V_flag = (t >> 31) != 0 && (accwide >> 31) != -1;
}
```

```
/* MPY */
/*when has_dsp==true
```

Assembly Code Example

```
MPY r1,r2,r3      ; Multiply r2 by r3
MPY_S r1,r2        ; and put the least-significant
                   ; word of the result in r1
                   ; Multiply r1 by r2 and put low
                   ; part of result in r1
```

Syntax and Encoding

Instruction Code

MPY<.f>	a,b,c	00100bbb00011010FBBBCCCCC
MPY<.f>	a,b,u6	00100bbb01011010FBBBuuuuuu
MPY<.f>	b,b,s12	00100bbb10011010FBBBssssss
MPY<.cc><.f>	b,b,c	00100bbb11011010FBBBCCCCC
MPY<.cc><.f>	b,b,u6	00100bbb11011010FBBBuuuuuu
MPY_S	b,b,c	01111bbbccc01100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPY_S

Function

Signed 32 x 32 multiplication. Return the least-significant 32 bits of the result. For the non-DSP version, see [MPY](#), [MPY_S](#).

Extension Group

MPY_OPTION > 1 or has_dsp==true

Operation

```
t = b * c;
b = t.w0;
```

Instruction Format

op b, b, c

Syntax Example

MPY_S b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Return the least-significant 32 bits of the product in the destination register. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers          /* MPY_S */
t  = b * c;
b  = (t & 0xffff_ffff);
```

Assembly Code Example

```
MPY_S r1,r2                ; Signed multiplication of r1 an  
                           ;r2, and return the result in r2
```

Syntax and Encoding

Instruction Code

MPY_S	b, c	01111bbbccc01100

MPYD

Function

Signed 32x32 multiplication. Return a 64-bit result.

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```

Instruction Format

op A, b, c

Syntax Example

MPYD <.f> A,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	= Cleared

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Write the product to the wide accumulator. Return the least-significant 64 bits of the wide accumulator in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYD*/
accwide  = b * c;
ACC0_HI  = (accwide >> 32) & 0xffff_ffff;
ACC0_LO  = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = (accwide & 0xffff_ffff_ffff_ffff);
if F {
    N_flag = accwide < 0;
    V_flag = 0;
}
```

Assembly Code Example

```
MPYD r0,r2,r3      ; 32x32 multiplication of r2 and r3. The result is
                   ;stored in (r1, r0).
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
MPYD<.f>	a,b,c	00101bbb00011000FBBBCCCCC0AAAAAA
MPYD<.f>	a,b,u6	00101bbb01011000FBBBuuuuuuAAAAAA
MPYD<.f>	b,b,s12	00101bbb10011000FBBBssssssSSSSSS
MPYD<.cc><.f>	b,b,c	00101bbb11011000FBBBCCCCC0QQQQQ
MPYD<.cc><.f>	b,b,u6	00101bbb11011000FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

MPYDF

Function

Signed 32x32 fractional multiplication. Return a 64-bit result.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide = b * c;
    a = SAT64(accwide<<1);
}
```



Note

A is a register pair representing a 64-bit operand.

Instruction Format

op A, b, c

Syntax Example

MPYDF <.f> A,b,c

Timing Characteristics

Two cycle

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands, and write the product to the wide accumulator. Assign the saturated least 64-bits of the wide accumulator to the destination register pair.

The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYDF*/
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD !=
-1 : ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide, 63, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD !=
-1 : ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(accwide << 1, 63, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = (accwide >> 32) & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
}
```

Assembly Code Example

```
MPYDF r0,r2,r3    ; 32x32 signed fractional multiplication of r2 and r3.
                  The result is ;saturated and stored in (r1, r0).
```

Syntax and Encoding

Instruction Code		
MPYDF<.f>	a, b, c	00110bbb00010010FBBBCCCCCAAAAA
MPYDF<.f>	a, b, u6	00110bbb01010010FBBBuuuuuuAAAAA
MPYDF<.f>	b, b, s12	00110bbb10010010FBBBssssssSSSSSS
MPYDF<.cc><.f>	b, b, c	00110bbb11010010FBBBCCCCC0QQQQQ
MPYDF<.cc><.f>	b, b, u6	00110bbb11010010FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYDU

Function

Unsigned 32x32 multiplication with a 64-bit result.

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```

Instruction Format

op A, b, c

Syntax Example

MPYDU <.f> A,b,c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V	•	= Cleared

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand 1 and operand 2. Write the product to the wide accumulator. Return the least-significant 64 bits of the wide accumulator in the destination register pair.

Any flag updates occur only if the set flags suffix (.F) is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```
// note: accumulators modeled as 128b integers          /* MPYDU*/
accwide  = b * c;
ACC0_HI  = (accwide >> 32) & 0xffff_ffff;
ACC0_LO  = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = (accwide & 0xffff_ffff_ffff_ffff);
if F {
    V_flag = 0;
}
```

Assembly Code Example

```
MPYDU r0,r2,r3    ; 32x32 unsigned multiplication of r2 and r3. The
                  ;result is stored in (r1,r0)
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

Instruction Code

MPYDU<.f>	a, b, c	00101bbb00011001FBBBCCCCC AAAAAA
MPYDU<.f>	a, b, u6	00101bbb01011001FBBBuuuuuu AAAAAA
MPYDU<.f>	b, b, s12	00101bbb10011001FBBBssssss SSSSSS
MPYDU<.cc><.f>	b, b, c	00101bbb11011001FBBBCCCCC 0QQQQQ
MPYDU<.cc><.f>	b, b, u6	00101bbb11011001FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYF

Function

Signed 32x32 fractional multiplication.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide = b * c;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands, and write the product to the wide accumulator. Return the truncated and saturated second 32 bits of the wide accumulator to the destination register.

The sticky flag `DSP_CTRL.SAT` is set, regardless of the `.F` suffix, if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MPYF*/
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -
1 : ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(accwide >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        ACC0_GHI.V      = sat_en(b * c, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -
1 : ACC0_GHI.GUARD != 0;
    else
        ACC0_GHI.V = sat(b * c, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(accwide >> 31, 31, &a);
DSP_CTRL.SAT |= s || v | ACC0_GHI.V;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a == 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MPYF r0,r1,r2      ; Signed fractional multiplication of r2 and r3. The
                   ;result is saturated and stored in r0.

```

Syntax and Encoding

Instruction Code

MPYF<.f>	a,b,c	00110bbb00001010FBBBCCCCCAAAAAA
MPYF<.f>	a,b,u6	00110bbb01001010FBBBuuuuuuAAAAAA

MPYF<.f>	b,b,s12	00110 bbb 10 001010 F BBB ssssss SSSSSS
MPYF<.cc><.f>	b,b,c	00110 bbb 11 001010 F BBB CCCCC 0QQQQQ
MPYF<.cc><.f>	b,b,u6	00110 bbb 11 001010 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYFR

Function

Signed 32x32 fractional multiplication. The result is rounded.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide = b * c;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands, and write the product to the wide accumulator. Return the rounded and saturated second 32 bits of the wide accumulator to the destination register.

The sticky flag DSP_CTRL.SAT is set, regardless of the .F suffix, if the result saturates. The other flags are updated only if the .F suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MPYFR*/
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -
1 : ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -
1 : ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(round(accwide,31) >> 31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = (accwide >> 32) & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a == 0;
            N_flag = a < 0;
            V_flag = s;
        }

```

Assembly Code Example

```

MPYFR r0,r1,r2    ; Signed fractional multiplication of r2 and r3. The
                  ;result is saturated, rounded, and stored in r0.

```

Syntax and Encoding

Instruction Code

MPYFR<.f>	a,b,c	00110bbb00001011FBBBCCCCCAAAAAA
MPYFR<.f>	a,b,u6	00110bbb01001011FBBBuuuuuuAAAAAA

MPYFR<.f>	b,b,s12	00110 bbb 10 001011 F BBB ssssss SSSSSS
MPYFR<.cc><.f>	b,b,c	00110 bbb 11 001011 F BBB CCCCC 0QQQQQ
MPYFR<.cc><.f>	b,b,u6	00110 bbb 11 001011 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYM MPYH

Function

32 x 32 Signed Multiply High For the non-DSP version, see [MPYM MPYH](#).

 **Note**

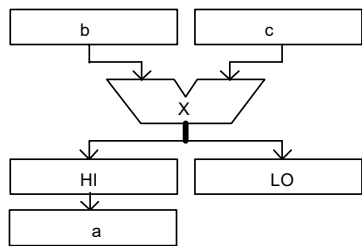
The MPYH mnemonic is deprecated.

Extension Group

- MPY_OPTION > 1 or has_dsp==true

Operation

if (cc) a = (signed) (b * c) >> 32;



Instruction Format

op a, b, c

Syntax Example

MPYM<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set is the accumulator is negative
C		= Unchanged
V	•	= Always cleared.

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Return bits [63:32] of the result.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers          /* MPYM */
w  = b * c;                                     /*when has_dsp==true
a  = (w >> 32 & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = a < 0;
    V_flag = 0;
}
```

Assembly Code Example

```
MPYM r1,r2,r3      ;Multiply r2 by r3 and put high part of the result in
                   ;r1
```

Syntax and Encoding

Instruction Code

MPYM<.f>	a,b,c	00100bbb00011011FBBBCCCCC0AAAAAA
MPYM<.f>	a,b,u6	00100bbb01011011FBBBuuuuuuAAAAAA
MPYM<.f>	b,b,s12	00100bbb10011011FBBBssssssSSSSSS
MPYM<.cc><.f>	b,b,c	00100bbb11011011FBBBCCCCC0QQQQQ
MPYM<.cc><.f>	b,b,u6	00100bbb11011011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYMU MPYHU

Function

32 x 32 Unsigned integer Multiply High. For the non-DSP version, see [MPYMU MPYHU](#).



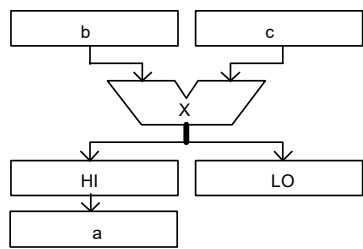
Note The MPYHU mnemonic is deprecated.

Extension Group

- MPY_OPTION > 1 or has_dsp==true

Operation

if (cc) a = (unsigned) (b * c) >> 32;dest (src1 X src2).high



Instruction Format

op a, b, c

Syntax Example

MPYMU<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared.
C		= Unchanged
V	•	= Always cleared.

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand 1 and operand 2. Return bits [63:32] of the result.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers          /* MPYMU */
t = (unsigned) b * c;                          /* when has_dsp==true */
a = (t >> 32 & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = 0;
    V_flag = 0;
}
```

Assembly Code Example

```
MPYMU r1,r2,r3      ;Multiply r2 by r3 and put high part of the result
                    ;in r1
```

Syntax and Encoding

Instruction Code

MPYMU<.f>	a,b,c	00100bbb00011100FBBBCCCCCAAAAAA
MPYMU<.f>	a,b,u6	00100bbb01011100FBBBuuuuuuAAAAAA
MPYMU<.f>	b,b,s12	00100bbb10011100FBBBssssssSSSSSS
MPYMU<.cc><.f>	b,b,c	00100bbb11011100FBBBCCCCC0QQQQQ
MPYMU<.cc><.f>	b,b,u6	00100bbb11011100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYU

Function

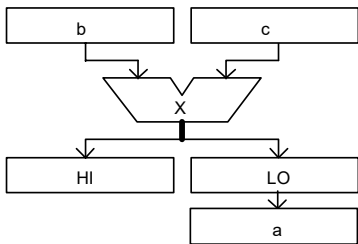
32 x 32 Unsigned Multiply, returning least-significant 32-bit word of result. For the non-DSP version, see [MPYU](#).

Extension Group

(MPY_OPTION > 1) or (has_dsp == true)

Operation

if (cc) a = (unsigned) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPYU<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Set when the result cannot be represented as a 32-bit unsigned integer

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand 1 and operand 2. Return the least-significant bits of the product to the destination register.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers
t = (unsigned)b * c;
a = (t & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = 0;
    V_flag = (t >> 32) != 0;
}
```

```
/* MPYU */
/* when has_dsp==true
```

Assembly Code Example

```
MPYU r1,r2,r3    ;Multiply r2 by r3 and put low part of the result in
                  ;r1
```

Syntax and Encoding

Instruction Code

MPYU<.f>	a,b,c	00100 bbb 00 011101 FBBB CCCCC AAAAAA
MPYU<.f>	a,b,u6	00100 bbb 01 011101 FBBB uuuuuu AAAAAA
MPYU<.f>	b,b,s12	00100 bbb 10 011101 FBBB ssssss SSSSSS
MPYU<.cc><.f>	b,b,c	00100 bbb 11 011101 FBBB CCCCC 0QQQQQ
MPYU<.cc><.f>	b,b,u6	00100 bbb 11 011101 FBBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYUW

Function

16 x 16 unsigned Multiply, return 32-bit word result. For the non-DSP version, see [MPYUW](#), [MPYUW_S](#).

Extension Group

(MPY_OPTION > 0) or (has_dsp == true)

Operation

For has_dsp==true

```
if (cc) {
    accwide = (unsigned) b.h0 * c.h0;
    a = accwide.w0;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYUW<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Always cleared

Description

Perform an unsigned 16-bit by 16-bit multiply of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register. The 16-bit source operands use the lower 16 bits of the source registers. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
// note: data modeled as 128b integers
t = (unsigned)b.h0 * c.h0;
a = (t & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = 0;
    V_flag = 0;
}
```

```
/* MPYUW */
/* when has_dsp==true
```

Assembly Code Example

```
MPYUW r1,r2, r3    ; Multiply r2 by r3 and put the result in r1
```

Syntax and Encoding

Instruction Code

MPYUW<.f>	a,b,c	00100bbb00011111FBBBCCCCC	AAAAAA
MPYUW<.f>	a,b,u6	00100bbb01011111FBBBuuuuuu	AAAAAA
MPYUW<.f>	b,b,s12	00100bbb10011111FBBBssssss	SSSSSS
MPYUW<.cc><.f>	b,b,c	00100bbb11011111FBBBCCCCC	0QQQQQ
MPYUW<.cc><.f>	b,b,u6	00100bbb11011111FBBBuuuuuu	1QQQQQ
MPYUW_S	b,b,c	01111bbbccc	01010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYUW_S

Function

Unsigned 16 x 16 multiplication, return a 32-bit result. For the non-DSP version, see [MPYUW](#), [MPYUW_S](#).

Extension Group

(MPY_OPTION > 0) or (has_dsp == true)

Operation

```
if (cc) {
    t = (unsigned) b.h0 * c.h0;
    b = t.w0;
}
```

Instruction Format

op b, c

Syntax Example

MPYUW_S<.f> b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an unsigned 16-bit by 16-bit multiply of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register. The 16-bit source operands use the lower 16 bits of the source registers. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers                                /* MPYUW_S */
t = (unsigned)b.h0 * c.h0;
b = (t & 0xffff_ffff);
```

Assembly Code Example

```
MPYUW_S r1, r2    ; Unsigned 16x16 multiplication of r2 and r1 and  
                  ;return the result in r1
```

Syntax and Encoding

Instruction Code

MPYUW_S	b, c	01111bbbccc01010

MPYW

Function

16 x 16 Signed Multiply. For the non-DSP version, see [MPYW](#), [MPYW_S](#).

Extension Group

(MPY_OPTION > 0) or (has_dsp == true)

Operation

For has_dsp==true

```
if (cc) {
    accwide = b.h0 * c.h0;
    a = accwide.w0;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYW<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 32-bit result is set
C		= Unchanged
V	•	= Set when the unsigned result overflows

Description

Perform a signed 16-bit by 16-bit multiply of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register. The 16-bit source operands use the lower 16 bits of the source registers. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
// note: accumulators modeled as 128b integers      /* MPYW */
t  = b.h0 * c.h0;                                   /* when has_dsp==true
a  = (t & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = t < 0;
    V_flag = (t >> 32) != 0;
}
```

Assembly Code Example

```
MPYW r1,r2, r3    ; Multiply r2 by r3 and put the result in r1
```

Syntax and Encoding

Instruction Code

MPYW<.f>	a,b,c	00100 bbb 00 01111 0 FBBB CCCCC AAAAAA
MPYW<.f>	a,b,u6	00100 bbb 01 01111 0 FBBB uuuuuu AAAAAA
MPYW<.f>	b,b,s12	00100 bbb 10 01111 0 FBBB ssssss SSSSSS
MPYW<.cc><.f>	b,b,c	00100 bbb 11 01111 0 FBBB CCCCC 0QQQQQ
MPYW<.cc><.f>	b,b,u6	00100 bbb 11 01111 0 FBBB uuuuuu 1QQQQQ
MPYW_S	b,b,c	01111 bbb ccc 01001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYW_S

Function

Signed 16 x 16 multiplication, whole 32-bit word result. For the non-DSP version, see [MPYW](#), [MPYW_S](#).

Extension Group

(MPY_OPTION > 0) or (has_dsp == true)

Operation

```
t = b.h0 * c.h0;
b = t.w0;
```

Instruction Format

op b, c

Syntax Example

MPYW_S<.f> b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed 16-bit by 16-bit multiply of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register. The 16-bit source operands use the lower 16 bits of the source registers. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers                                /* MPYW_S */
t  = b.h0 * c.h0;
b  = (t & 0xffff_ffff);
```

Assembly Code Example

```
MPYW_S r1, r2      ; Signed 16x16 multiplication of r2 and r1 and  
                   ;return the result in r1
```

Syntax and Encoding

Instruction Code

MPYW_S	b, c	01111bbbccc01001

MPYWHFL

Function

Signed fractional multiplication of the 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h0 <<16;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHFL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of the 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the saturated most-significant 32 bits of the wide accumulator without the guard bits to the destination operand. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c.h0 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h0 << 16, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(accwide>>31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
/* MPYWHFL
*/
```

Assembly Code Example

```
MPYWHFL r0,r2,r3      ; Signed fractional low multiply r2 and r3 and
                      ; return result in register r0
```

Syntax and Encoding

Instruction Code

MPYWHFL<.f>	a,b,c	00110bbb00100100FBBBCCCCCAAAAAA
MPYWHFL<.f>	a,b,u6	00110bbb01100100FBBBuuuuuuAAAAAA

MPYWHFL<.f>	b, b, s12	00110bbb10100100FBBBssssssSSSSSS
MPYWHFL<.cc><.f>	b, b, c	00110bbb11100100FBBBCCCCC0QQQQQ
MPYWHFL<.cc><.f>	b, b, u6	00110bbb11100100FBBBuuuuuu100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHFLR

Function

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. The result is saturated and rounded.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h0 <<16;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHFLR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the saturated and rounded most-significant 32 bits of the wide accumulator without the guard bits to the destination operand. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYWHFLR
s = 0;                                                                    */
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c.h0 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h0 << 16, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(round(accwide, 31) >> 31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
```

Assembly Code Example

```
MPYWHFLR r0,r2,r3      ; Signed fractional low multiply r2 and r3 and
                        ; return the saturated and rounded result in
                        ;register r0
```

Syntax and Encoding

Instruction Code

```
MPYWHFLR<.f>          a,b,c          00110bbb00100101FBBBCCCCCAAAAA
```


MPYWHFLR<.f>	a,b,u6	00110bbb01100101FBBBuuuuuuAAAAAA
MPYWHFLR<.f>	b,b,s12	00110bbb10100101FBBBssssssSSSSSS
MPYWHFLR<.cc><.f>	b,b,c	00110bbb11100101FBBBCCCCC000000
MPYWHFLR<.cc><.f>	b,b,u6	00110bbb11100101FBBBuuuuuu100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHFM

Function

Signed fractional multiplication of 32 bits of one operand and most-significant 16 bits of the other operand.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h1 <<16;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHFM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the saturated most-significant 32 bits of the wide accumulator without the guard bits to the destination operand. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c.h1 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h1 << 16, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(accwide>>31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
/* MPYWHFM
*/
```

Assembly Code Example

```
MPYWHFM r0,r2,r3      ; Signed fractional high multiply r2 and r3 and
                      ; return result in register r0
```

Syntax and Encoding

Instruction Code

MPYWHFM<.f>	a,b,c	00110bbb00100000FBBBCCCCCAAAAA
MPYWHFM<.f>	a,b,u6	00110bbb01100000FBBBuuuuuuAAAAA

MPYWHFM<.f>	b, b, s12	00110bbb10100000FBBBssssssSSSSSS
MPYWHFM<.cc><.f>	b, b, c	00110bbb11100000FBBBCCCCC0QQQQQ
MPYWHFM<.cc><.f>	b, b, u6	00110bbb11100000FBBBuuuuuu100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHFMR

Function

Signed fractional multiplication of 32 bits of one operand and most-significant 16 bits of the other operand. The result is rounded and saturated.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h1 <<16;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHFMR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the saturated most-significant 32 bits of the wide accumulator without the guard bits to the destination operand. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYWHFMR
s = 0;                                                                    */
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        ACC0_GHI.V      = sat(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V = sat(b * c.h1 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h1 << 16, 63, &accwide);
            v = sat(round(accwide, 31) >> 31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
```

Assembly Code Example

```
MPYWHFMR r0,r2,r3      ; Signed fractional high multiply r2 and r3 and
                        ; return result in register r0
```

Syntax and Encoding

Instruction Code

MPYWHFMR<.f>	a,b,c	00110bbb00100001FBBBCCCCCAAAAAA
MPYWHFMR<.f>	a,b,u6	00110bbb01100001FBBBuuuuuuAAAAAA
MPYWHFMR<.f>	b,b,s12	00110bbb10100001FBBBsssssssSSSSSS

MPYWHFMR<.cc><.f>	b, b, c	00110	bbb11	100001	FBBB	CCCCC	000000
MPYWHFMR<.cc><.f>	b, b, u6	00110	bbb11	100001	FBBB	uuuuuu	100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHL

Function

Signed multiplication of 32 bits of one operand and least-significant 16 bits of the other operand.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h0;
    a = accwide;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	=Set if the accumulator cannot be represented as a 32-bit signed integer

Description

Signed multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the least-significant 32 bits of the wide accumulator. The other flags are updated only if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide = b * c.h0;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
    s = accwide >> 31 != 0 && accwide >> 31 != -1
else
    accwide = b * c.h0;
    s = accwide >> 31 != 0 && accwide >> 31 != -1
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    Z_flag = a == 0;
    N_flag = accwide < 0;
    V_flag |= s;
}
```

Assembly Code Example

```
MPYWHL r0,r2,r3      ; Signed integer multiplication r2 and r3 and
                     ; return result in register r0
```

Syntax and Encoding

Instruction Code		
MPYWHL<.f>	a, b, c	00110bbb000111100FBBBCCCCC AAAAAA
MPYWHL<.f>	a, b, u6	00110bbb010111100FBBBuuuuuu AAAAAA
MPYWHL<.f>	b, b, s12	00110bbb100111100FBBBssssss SSSSSS
MPYWHL<.cc><.f>	b, b, c	00110bbb110111100FBBBCCCCC 0QQQQQ
MPYWHL<.cc><.f>	b, b, u6	00110bbb110111100FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

MPYWHUL

Function

Unsigned multiplication of 32 bits of one operand and the least-significant 16 bits of the other operand.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = (unsigned) (b * c.h0);
    a = accwide;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHUL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register value is zero
N	•	= Cleared
C		= Unchanged
V	•	=Set if the accumulator cannot be represented as a 32-bit unsigned integer

Description

Unsigned multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the least-significant 32 bits of the wide accumulator. The other flags are updated only if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide = b * c.h0;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
    s = accwide >> 32 != 0;
else
    accwide = b * c.h0;
    s = accwide >> 32 != 0;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    Z_flag = a == 0;
    N_flag = a < 0;
    V_flag |= s;
}
```

/* MPYWHUL
*/

Assembly Code Example

```
MPYWHUL r0,r2,r3      ; Unsigned integer multiplication r2 and r3 and
                      ; return result in register r0
```

Syntax and Encoding

Instruction Code		
MPYWHUL<.f>	a, b, c	00110bbb000111110FBBBCCCCC AAAAAA
MPYWHUL<.f>	a, b, u6	00110bbb010111110FBBBuuuuuu AAAAAA
MPYWHUL<.f>	b, b, s12	00110bbb100111110FBBBssssss SSSSSS
MPYWHUL<.cc><.f>	b, b, c	00110bbb110111110FBBBCCCCC 0QQQQQ
MPYWHUL<.cc><.f>	b, b, u6	00110bbb110111110FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

MPYWHKL

Function

Signed integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    acchi = b * c.h0 >> 16;
    a = acchi;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHKL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	=Set if the accumulator cannot be represented by a 32-bit signed integer

Description

Signed integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator. Return the least-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MPYWHKL */
if (DSP_CTRL.GE)
    acchi = (b * c.h0) >> 16;
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
else
    acchi = (b * c.h0) >> 16;
    ACC0_HI = acchi & 0xffff_ffff;
    a = acchi & 0xffff_ffff;
    if (DSP_CTRL.GE)
        rem = acchi >> 39;
    else
        rem = acchi >> 31;
    if F {
        N_flag = rem & 1;
        V_flag = rem != -1 && rem != 0;
        Z_flag = a == 0;
    }

```

Assembly Code Example

```

MPYWHKL r0,r2,r3                ; Signed integer multiplication of r2 and r3
                                ; shift result by 16 bits, and return the shifted
                                ; result in register r0

```

Syntax and Encoding

Instruction Code

MPYWHKL<.f>	a, b, c	00110bbb00101010FBBBCCCCC	AAAAAA
MPYWHKL<.f>	a, b, u6	00110bbb01101010FBBBuuuuuu	AAAAAA
MPYWHKL<.f>	b, b, s12	00110bbb10101010FBBBssssss	SSSSSS
MPYWHKL<.cc><.f>	b, b, c	00110bbb11101010FBBBCCCCC	0QQQQQ
MPYWHKL<.cc><.f>	b, b, u6	00110bbb11101010FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHKUL

Function

Unsigned integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    acchi = b * c.h0 >> 16;
    a = acchi;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHKUL <.f> a,b,c

STATUS32 Flags Affected

Z		= Set if the result is zero
N	•	= Cleared
C		= Unchanged
V	•	=Set if the accumulator cannot be represented as a 32-bit unsigned integer

Description

Unsigned integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator. Return the least-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYWHKUL */
if (DSP_CTRL.GE)
    acchi = (b * c.h0) >> 16;
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    s = acchi >> 40 != 0;
else
    acchi = (b * c.h0;) >> 16
    s = acchi >> 32 != 0;
ACC0_HI = acchi & 0xffff_ffff;
a = acchi & 0xffff_ffff;
if F {
    V_flag = s;
    N_flag = 0;
    Z_flag = a == 0;
}
```

Assembly Code Example

```
MPYWHKUL r0,r2,r3      ; Unsigned integer multiplication of r2 and r3
                       ; shift result by 16 bits, and return the shifted
                       ; result in register r0
```

Syntax and Encoding

Instruction Code

MPYWHKUL<.f>	a, b, c	00110bbb00101011FBBBCCCCC	AAAAAA
MPYWHKUL<.f>	a, b, u6	00110bbb01101011FBBBuuuuuu	AAAAAA
MPYWHKUL<.f>	b, b, s12	00110bbb10101011FBBBssssss	SSSSSS
MPYWHKUL<.cc><.f>	b, b, c	00110bbb11101011FBBBCCCCC	0QQQQQ
MPYWHKUL<.cc><.f>	b, b, u6	00110bbb11101011FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

MSUBDF

Function

Signed 32x32 fractional multiplication and subtraction; return a 64-bit result.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide -= b * c;
    A = SAT64(accwide<<1);
}
```

Instruction Format

op A, b, c

Syntax Example

MSUBDF <.f> A,b,c

Timing Characteristics

Two cycle

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Perform a signed fractional multiplication of 32 bits of operand *b* and operand *c*; subtract the product from the wide accumulator. Return the saturated least-significant 64 bits of the wide accumulator.

The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (`LP_COUNT/r60`) as a destination operand register.
- Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBDF
s = 0;                                                                    */
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide, 63, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
            ACC0_GHI.V = sat_en(accwide - b * c, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
            ACC0_GHI.V = sat(accwide - b * c, 63, &accwide);
            ACC0_GHI.G = 0;
            v |= sat(accwide << 1, 63, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = (accwide >> 32) & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }

```

Assembly Code Example

```

MSUBDF r0,r2,r3                ; Signed fractional multiplication of r2 and r3
                                ;and subtract the product from the accumulator
                                ; return result in register (r1,r0)

```

Syntax and Encoding

		Instruction Code
MSUBDF<.f>	a, b, c	00110bbb00010101FBBBCCCCC AAAAAA
MSUBDF<.f>	a, b, u6	00110bbb01010101FBBBuuuuuu AAAAAA
MSUBDF<.f>	b, b, s12	00110bbb10010101FBBBssssss SSSSSS
MSUBDF<.cc><.f>	b, b, c	00110bbb11010101FBBBCCCCC 000000
MSUBDF<.cc><.f>	b, b, u6	00110bbb11010101FBBBuuuuuu 100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBF

Function

Signed 32x32 fractional multiplication and subtraction; return a 32-bit result.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide -= b * c;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Perform a signed fractional multiplication of the 32 bits of the operands *b* and operand *c*; subtract the product from the wide accumulator. Return the saturated second 32-bit word of the wide accumulator. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBF */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide - b * c, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide - b * c, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MSUBF r1,r2,r3          ; Signed fractional multiplication of r2 and r3
                        ;and subtract the product from the accumulator
                        ; return result in register r1

```


Syntax and Encoding

		Instruction Code
MSUBF<.f>	a, b, c	00110bbb00001110FBBBCCCCC(AAAAAA)
MSUBF<.f>	a, b, u6	00110bbb01001110FBBBuuuuuu(AAAAAA)
MSUBF<.f>	b, b, s12	00110bbb10001110FBBBssssss(SSSSSS)
MSUBF<.cc><.f>	b, b, c	00110bbb11001110FBBBCCCCC(0QQQQQ)
MSUBF<.cc><.f>	b, b, u6	00110bbb11001110FBBBuuuuuu(1QQQQQ)

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBFR

Function

Signed 32x32 fractional multiplication and subtraction; return a rounded and saturated 32-bit result.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide -= b * c;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Perform a signed fractional multiplication of the 32 bits of the operands *b* and operand *c*; subtract the product from the wide accumulator. Return the rounded and saturated second 32-bit word of the wide accumulator. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBFR
s = 0;                                                                    */
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
            ACC0_GHI.V = sat_en(accwide - b * c, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
            ACC0_GHI.V = sat(accwide - b * c, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(round(accwide,31) >> 31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = (accwide >> 32) & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }

```

Assembly Code Example

```

MSUBFR r1,r2,r3                ; Signed fractional multiplication of r2 and r3 and
                                ;subtract the product from the accumulator
                                ; return rounded and saturated result in register r1

```

Syntax and Encoding

		Instruction Code
MSUBFR<.f>	a, b, c	00110bbb00001111FBBBCCCCC(AAAAAA)
MSUBFR<.f>	a, b, u6	00110bbb01001111FBBBuuuuuu(AAAAAA)
MSUBFR<.f>	b, b, s12	00110bbb10001111FBBBssssss(SSSSSS)
MSUBFR<.cc><.f>	b, b, c	00110bbb11001111FBBBCCCCC(0QQQQQ)
MSUBFR<.cc><.f>	b, b, u6	00110bbb11001111FBBBuuuuuu(1QQQQQ)

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBWHFL

Function

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide -= b * c.h0 <<32;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBWHFL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBWHFL */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - b * c.h0 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.G = 0;
        ACC0_GHI.V = sat(accwide - b * c.h0 << 16, 63, &accwide); v =
sat(accwide>>31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MSUBWHFL r0,r2,r3          ; Signed fractional multiply-subtract r2 and r3
                           ;and return result in register r0

```

Syntax and Encoding

		Instruction Code
MSUBWHFL<.f>	a, b, c	00110bbb00010100FBBBCCCCC AAAAAA
MSUBWHFL<.f>	a, b, u6	00110bbb01010100FBBBuuuuuu AAAAAA
MSUBWHFL<.f>	b, b, s12	00110bbb10010100FBBBssssss SSSSSS
MSUBWHFL<.cc><.f>	b, b, c	00110bbb11010100FBBBCCCCC 0QQQQQ
MSUBWHFL<.cc><.f>	b, b, u6	00110bbb11010100FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBWHFLR

Function

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the rounded and saturated result.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide -= b * c.h0 << 32;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBWHFLR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the rounded and saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - b * c.h0 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - b * c.h0 << 16, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide, 31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MSUBWHFLR r0,r2,r3      ; Signed fractional multiply-subtract r2 and r3
                        ;and return result in register r0

```

Syntax and Encoding

		Instruction Code
MSUBWHFLR<.f>	a, b, c	00110bbb00011010FBBBCCCCC AAAAAA
MSUBWHFLR<.f>	a, b, u6	00110bbb01011010FBBBuuuuuu AAAAAA
MSUBWHFLR<.f>	b, b, s12	00110bbb10011010FBBBssssss SSSSSS
MSUBWHFLR<.cc><.f>	b, b, c	00110bbb11011010FBBBCCCCC 0QQQQQ
MSUBWHFLR<.cc><.f>	b, b, u6	00110bbb11011010FBBBuuuuuu 100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBWHFM

Function

Signed fractional multiplication and subtraction of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide -= b * c.h1 <<32;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBWHFM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBWHFM */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - b * c.h1 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - b * c.h1 << 16, 63, &accwide); v =
sat(accwide>>31, 31, &a);
        ACC0_GHI.G = 0;
    DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
    ACC0_HI = accwide >> 32 & 0xffff_ffff;
    ACC0_LO = accwide & 0xffff_ffff;
    ACC0_GHI.Z = accwide == 0;
    ACC0_GHI.N = accwide < 0;
    if F {
        Z_flag = a = 0;
        N_flag = a < 0;
        V_flag = v;
    }

```

Assembly Code Example

```

MSUBWHFM r0,r2,r3      ; Signed fractional multiply-subtract r2 and r3 and
                        ; return the rounded and saturated result in register r0

```

Syntax and Encoding

		Instruction Code
MSUBWHFM<.f>	a, b, c	00110bbb00101100FBBBCCCCCAAAAA
MSUBWHFM<.f>	a, b, u6	00110bbb01101100FBBBuuuuuuAAAAA
MSUBWHFM<.f>	b, b, s12	00110bbb10101100FBBBssssssSSSSSS
MSUBWHFM<.cc><.f>	b, b, c	00110bbb11101100FBBBCCCCC0QQQQQ
MSUBWHFM<.cc><.f>	b, b, u6	00110bbb11101100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBWHFMR

Function

Signed fractional multiplication and subtraction of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand; with rounded result

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide -= b * c.h1 << 32;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBWHFMR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the rounded and saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* MSUBWHFMR */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - b * c.h1 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - b * c.h1 << 16, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide, 31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MSUBWHFMR r0,r2,r3      ; Signed fractional multiply-subtract r2 and r3 and
                        ; return the rounded and saturated result in register r0

```

Syntax and Encoding

		Instruction Code
MSUBWHFMR<.f>	a, b, c	00110bbb00101101FBBBCCCCCAAAAAA
MSUBWHFMR<.f>	a, b, u6	00110bbb01101101FBBBuuuuuuAAAAAA
MSUBWHFMR<.f>	b, b, s12	00110bbb10101101FBBBssssssSSSSSS
MSUBWHFMR<.cc><.f>	b, b, c	00110bbb11101101FBBBCCCCC0QQQQQ
MSUBWHFMR<.cc><.f>	b, b, u6	00110bbb11101101FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

NEGS

Function

Negate a 32-bit word and saturate the result.

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT32}(-c);$$

Instruction Format

op b,c

Syntax Example

NEGS<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if input is 0x8000_0000; otherwise cleared

Description

Negate long word operand, c, and place the result in the destination register b while saturating. An absolute source value of 0x8000_0000 yields 0x7FFF_FFFF. The saturation flag, DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. If the set flags suffix (.F) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = (c == 0x8000_0000 ? 0x7fff_ffff : -c);          /* NEGS */
DSP_CTRL.SAT |= c == 0x8000_0000;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = b < 0;
    STATUS32.V = c == 0x8000_0000;
}
```

Assembly Code Example

```
NEGS r1,r2    ; Negate r2 and write saturated result into r1
```

Syntax and Encoding

Instruction Code

NEGS<.f>	b,c	00101 bbb 00 000111 F BBB CCCCC 000111
NEGS<.f>	b,limm	00101 bbb 00 101111 F BBB 111110 000111
NEGS<.f>	0,c	00101 110 00 101111 F 111 CCCCC 000111
NEGS<.f>	0,limm	00101 110 00 101111 F 111 111110 000111
NEGS<.f>	b,u6	00101 bbb 01 000111 F BBB uuuuuu 000111
NEGS<.f>	0,u6	00101 110 01 101111 F 111 uuuuuu 000111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

NEGSH

Function

Negate the lower 16 bits of a word and saturate the result.

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT16}(-c);$$

Instruction Format

op b,c

Syntax Example

NEGSH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if input is 0x8000_0000; otherwise cleared

Description

Negate lower 16 bits of the operand, c, and place the result in the destination register b while saturating. An absolute source value of 0x8000_0000 yields 0x7FFF_FFFF. The saturation flag, DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. If the set flags suffix (.F) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
t = c & 0x0000_ffff; /* NEGSH */
b = (t == 0x0000_8000 ? 0x00007fff : -t);
DSP_CTRL.SAT |= t == 0x0000_8000;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = b < 0;
    STATUS32.V = t == 0x0000_8000;
}
```

Assembly Code Example

```
NEGSH r1,r2    ; Negate lower 16 bits of r2 and write saturated result
               ;into r1
```

Syntax and Encoding

Instruction Code

NEGSH<.f>	b,c	00101 bbb 00 101111 FBBB CCCCC 000110
NEGSH<.f>	b,limm	00101 bbb 00 101111 FBBB 111110 000110
NEGSH<.f>	0,c	00101 110 00 101111 F111 CCCCC 000110
NEGSH<.f>	0,limm	00101 110 00 101111 F111 111110 000110
NEGSH<.f>	0,u6	00101 110 01 101111 F111 uuuuuu 000110
NEGSH<.f>	b,u6	00101 bbb 01 101111 FBBB uuuuuu 000110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

NORMACC

Function

Normalize the accumulator.

Extension Group

DSP Accumulator

Operation

```
switch (c.b1) {
0: b.b0 = norm(accwide);
1: b.b0 = norm(acclo);
2: b.b0 = norm(acchi);
3: b.b2 = norm(acchi); b.b0 = norm(acclo);
}
```

Instruction Format

op b,c

Syntax Example

NORMACC b, c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Return the left shift amount for normalizing an accumulator to the fractional dot. C operand consists of one field: the accumulator select. Return the result in B. B is a signed byte and is in the range $-8 \leq B \leq 64$. The result is a vector in case the vector accumulator is selected.

Pseudo Code

```

short norm_var(int128 c, int w) {                                     /* NORMACC */
    int i;
    for (i = 0; (i < w) && ((c >> i) != 0) && ((c >> i) != -1); i++);
    return w - i - 1;
}
switch (c.b1) {
    case 0:
        if (DSP_CTRL.GE)
            accwide = ACC0_GHI.GUARD << 64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            if (DSP_CTRL.PA)
                B = norm_var(accwide, 72)-8;
            else
                B = norm_var(accwide, 72)-9;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            if (DSP_CTRL.PA)
                B = norm_var(accwide, 64);
            else
                B = norm_var(accwide, 64)-1;
        B &= 0xff;
        break;
    case 1:
        if (DSP_CTRL.GE)
            acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
            if (DSP_CTRL.PA)
                B = norm_var(acclo, 40)-8;
            else
                B = norm_var(acclo, 40)-9;
        else
            acclo = ACC0_LO;
            if (DSP_CTRL.PA)
                B = norm_var(acclo, 32);
            else
                B = norm_var(acclo, 32)-1;
        B &= 0xff;
        break;
    case 2:
        if (DSP_CTRL.GE)
            acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
            if (DSP_CTRL.PA)
                B = norm_var(acchi, 40)-8;
            else
                B = norm_var(acchi, 40)-9;
        else
            acchi = ACC0_HI;
            if (DSP_CTRL.PA)
                B = norm_var(acchi, 32);
            else
                B = norm_var(acchi, 32)-1;
        B &= 0xff;
        break;
}

```

```

case 3:
  if (DSP_CTRL.GE)
    acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
    if (DSP_CTRL.PA)
      BL = norm_var(acclo, 40)-8;
    else
      BL = norm_var(acclo, 40)-9;
  else
    acclo = ACC0_LO;
    if (DSP_CTRL.PA)
      BL = norm_var(acclo, 32);
    else
      BL = norm_var(acclo, 32)-1;
  if (DSP_CTRL.GE)
    acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
    if (DSP_CTRL.PA)
      BH = norm_var(acchi, 40)-8;
    else
      BH = norm_var(acchi, 40)-9;
  else
    acchi = ACC0_HI;
    if (DSP_CTRL.PA)
      BH = norm_var(acchi, 32);
    else
      BH = norm_var(acchi, 32)-1;
  B = (BH && 0x000000ff) << 16 | (BL & 0x000000ff);
  break;
}

```

Assembly Code Example

```

NORMACC r1,r2      ; Normalize the accumulator value based on fields in r2;
                   ;return result to r0

```

Syntax and Encoding

Instruction Code		
NORMACC	b,c	00101bbb001011110BBBCCCCC011001
NORMACC	b,limm	00101bbb001011110BBB111110011001
NORMACC	0,c	00101110001011110111CCCCC011001
NORMACC	0,limm	00101110001011110111111110011001
NORMACC	b,u6	00101bbb011011110BBBuuuuuu011001
NORMACC	0,u6	00101110011011110111uuuuuu011001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

REM

Function

2's complement integer Remainder. For the non-DSP version, see [REM](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

```
if (cc == true) {
  if ((c != 0) && ((b != 0x80000000) || c != 0xffffffff)) {
    r = b % c; // same sign as divisor
    q = b / c; // truncate to zero
    if (DSP_DIVSQRT == radix2) {
      ACC0_LO = r;
      ACC0_HI = q;
    }
    a = r;
    if (F == 1) {
      Z_flag = (a == 0) ? 1 : 0;
      N_flag = a < 0;
      V_flag = 0;
    }
  } else {
    if ((c == 0) && (STATUS32.DZ == 1)) {
      RaiseException (EV_DivZero);
    } else if (F == 1) {
      Z_flag = 0;
      N_flag = 0;
      V_flag = 1;
    }
  }
}
```

Instruction Format

op a, b, c

Syntax Example

REM <f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set

C		= Unchanged
V	•	= Set if divisor is zero or if an arithmetic overflow occurs

Description

If the divisor (c) is non-zero, the destination register is assigned the remainder obtained by signed integer division of source operand (b) by source operand (c). When the DSP_DIVSQRT radix-2 option is configured, the quotient is written to the high accumulator and the remainder is written in the low accumulator. The flags are only be updated if the .F suffix is used.

If the divisor (c) is zero or an overflow occurs, the destination register is never updated. If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised. An arithmetic overflow is signaled if the quotient of (b / c) cannot be represented in 32 bits. This overflow only occurs in one specific case, which is the division of the largest representable negative value (0x80000000 r 0xFFFFFFFF) by -1. If an arithmetic overflow occurs, the overflow flag (V) is set to 1.

See [Appendix Table A-3, “ISA Features: ARC EM and ARC HS”](#) for implementation specific details of this instruction when a divide-by-zero occurs and STATUS32.DZ bit is 0.

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If a division-by-zero is attempted when the EV_DivZero exception is disabled, or if the quotient cannot be represented in 32 bits, the overflow flag (V) is set to 1 and the destination register is cleared.

Pseudo Code

```

if (cc == true) {
    if ((c != 0) && ((b != 0x80000000) || c != 0xffffffff)) {
        r = b % c; // same sign as divisor
        q = b / c; // truncate to zero
        if (DSP_DIVSQRT == radix2) {
            ACC0_LO = r;
            ACC0_HI = q;
        }
        a = r;
        if (F == 1) {
            Z_flag = (a == 0) ? 1 : 0;
            N_flag = a < 0;
            V_flag = 0;
        }
    } else {
        if ((c == 0) && (STATUS32.DZ == 1)) {
            RaiseException (EV_DivZero);
        } else if (F == 1) {
            Z_flag = 0;
            N_flag = 0;
            V_flag = 1;
        }
    }
}
}

```

dsp_divsqrt!= none

Assembly Code Example

```
REM r1,r2,r3 ; r1 is assigned r2 % r3
```

Syntax and Encoding

Instruction Code

REM<.f>	a,b,c	00101bbb00001000FBBBCCCCCAAAAAA
REM<.cc><.f>	b,b,c	00101bbb11001000FBBBCCCCC0QQQQQ
REM<.cc><.f>	b,b,limm	00101bbb11001000FBBB1111100QQQQQ
REM<.cc><.f>	b,b,u6	00101bbb11001000FBBBuuuuuu1QQQQQ
REM<.f>	b,b,s12	00101bbb10001000FBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

`SUB.F 0, R0, R1`

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

REMU

Function

Unsigned integer Remainder. For the non-DSP version, see [REMU](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

```

if (cc == true) {
  if (c != 0) {
    q = b / c;
    r = b % c;
    if (DSP_DIVSQRT == radix2) {
      // accumulator only updated for radix-2
      ACC0_LO = r;
      ACC0_HI = q;
    }
    a = r;
    if (F == 1) {
      Z_flag = (a == 0) ? 1 : 0;
      N_flag = 0;
      V_flag = 0;
    }
  } else {
    if (STATUS32.DZ == 1) {
      RaiseException (EV_DivZero);
    } else if (F == 1) {
      Z_flag = 0;
      N_flag = 0;
      V_flag = 1;
    }
  }
}

```

Instruction Format

op a, b, c

Syntax Example

REMU <.f> a,b,c

STATUS32 Flags Affected

Z $\left[\begin{array}{|c|} \bullet \end{array} \right]$ = Set if result is zero

N	•	= Always cleared
C		= Unchanged
V	•	= Set if divisor is zero

Description

If the divisor (c) is non-zero, the destination register is assigned the remainder obtained by unsigned integer division of source operand (b) by source operand (c). When the DSP_DIVSQRT radix-2 option is configured, the quotient is written to the high accumulator and the remainder is written in the low accumulator. The flags are only be updated if the .F suffix is used.

If the divisor (c) is zero or an overflow occurs, the destination register is never updated. If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

See [Appendix Table A-3, “ISA Features: ARC EM and ARC HS”](#) for implementation specific details of this instruction when a divide-by-zero occurs and STATUS32.DZ bit is 0.

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If a division-by-zero is attempted when the EV_DivZero exception is disabled, or if the quotient cannot be represented in 32 bits, the overflow flag (V) is set to 1 and the destination register is cleared.

If the flag-update bit is set, an unsigned REMU operation always clears the N-flag.

Pseudo Code

```

if (cc == true) {
    if (c != 0) {
        q = b / c;
        r = b % c;
        if (DSP_DIVSQRT == radix2) {
            // accumulator only updated for radix-2
            ACC0_LO = r;
            ACC0_HI = q;
        }
        a = r;
        if (F == 1) {
            Z_flag = (a == 0) ? 1 : 0;
            N_flag = 0;
            V_flag = 0;
        }
    } else {
        if (STATUS32.DZ == 1) {
            RaiseException (EV_DivZero);
        } else if (F == 1) {
            Z_flag = 0;
            N_flag = 0;
            V_flag = 1;
        }
    }
}
}

```

dsp_divsqrt!= none

Assembly Code Example

```

REMU r1,r2,r3                ; r1 is assigned r2 % r3

```

Syntax and Encoding

Instruction Code

REMU<.f>	a,b,c	00101bbb00001001FBBBCCCCC0AAAAAA
REMU<.f>	a,b,u6	00101bbb01001001FBBBuuuuuuAAAAAA
REMU<.cc><.f>	b,b,c	00101bbb11001001FBBBCCCCC0QQQQQ
REMU<.cc><.f>	b,b,u6	00101bbb11001001FBBBuuuuuu1QQQQQ
REMU<.f>	b,b,s12	00101bbb10001001FBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

RNDH

Function

Round and saturate a 32-bit value to a 16-bit value

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT16}(\text{RND16}(c));$$

Instruction Format

op b,c

Syntax Example

RNDH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Set if the result saturates
V	•	= Saturation occurs when 0x7F_FFFF is rounded.

Description

Round a 32-bit input fractional value to a 16-bit fractional value. Saturate if rounding causes an overflow of the result. The saturation flag DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. Flag updates occur only if the set flags suffix (.F) is used.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
s = sat(round(c,16),31,&t); /* RNDH */
DSP_CTRL.SAT |= s;
b = (t>>16);
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b & 0x8000) == 0x8000;
    STATUS32.V = s;
}
```

Assembly Code Example

```
RNDH r1,r2    ; Round 32-bits in r2 and return saturated value into
               ;r1.
```

Syntax and Encoding

Instruction Code

RNDH<.f>	b,c	00101 bbb 00 101111 F BBB CCCCC 000011
RNDH<.f>	b,limm	00101 bbb 00 101111 F BBB 111110 000011
RNDH<.f>	0,c	00101 110 00 101111 F 111 CCCCC 000011
RNDH<.f>	0,limm	00101 110 00 101111 F 111 111110 000011
RNDH<.f>	b,u6	00101 bbb 01 101111 F BBB uuuuuu 000011
RNDH<.f>	0,u6	00101 110 01 101111 F 111 uuuuuu 000011

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SATH

Function

Saturate a 32-bit value to a 16-bit value

Extension Group

DSP basic arithmetic

Operation

$b = \text{SAT16}(c)$;

Instruction Format

op b,c

Syntax Example

SATH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of input is set
C		= Unchanged
V	•	= Set if input is smaller than 0xffff_8000 or bigger than 0x0000_7fff; otherwise cleared

Description

Saturate a 32-bit input fractional value to a 16-bit fractional value. Saturate if bit 15 up to 31 are not equal to the sign bit. The saturation flag DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. Flag updates occur only if the set flags suffix (.F) is used.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
s = sat(c, 15, &b);                                /* SATH
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = b < 0;
    STATUS32.V = s;
}
```

Assembly Code Example

```
SATH r1,r2    ; Saturate 32-bits in r2 and return saturated value
               ;into r1.
```

Syntax and Encoding

Instruction Code		
SATH<.f>	b,c	00101 bbb 00101111 FBBB CCCCC000010
SATH<.f>	b,limm	00101 bbb 00101111 FBBB 111110000010
SATH<.f>	0,c	00101 110 00101111 F111 CCCCC000010
SATH<.f>	0,limm	00101 110 00101111 F111 111110000010
SATH<.f>	b,u6	00101 bbb 01101111 FBBB uuuuuu000010
SATH<.f>	0,u6	00101 110 01101111 F111 uuuuuu000010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SATF

Function

Saturate a register based on the STATUS32.N and STATUS32.V flags.

Extension Group

DSP basic arithmetic

Operation

$$b = V ? (N ? 0x00000000 : 0xffffffff) : c;$$

Instruction Format

op b,c

Syntax Example

SATF<.f> b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

If the STATUS32.V flag is set, saturate the input operand based on the STATUS32.N flag value. If the STATUS32.N flag is set, saturate the input operand to a positive value, that is saturate the register with zeros (0s). If the STATUS32.N flag is not set, saturate the input operand to the negative value, that is saturate the register with ones (1s).

Pseudo Code

```

b = c                                     /* SATF
if (STATUS.V) {
    if (STATUS.N) {
        // saturate to positive
        b = 0x00000000;
    } else {
        // saturate to negative
        b = 0xffffffff;
    }
}

```

Assembly Code Example

```
SATF r0,r1    ; Saturate register r1 based on the N and V flags and
               ; return r0
```

Syntax and Encoding

Instruction Code

SATF<.f>	b, c	00101 bbb 00 101111 F BBB CCCCC 011010
SATF<.f>	b, u6	00101 bbb 01 101111 F BBB uuuuuu 011010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SBCS

Function

Signed subtraction with saturation and carry in

Extension Group

DSP basic arithmetic

Operation

$$a = \text{SAT32}((b - c) - \text{STATUS32.C});$$

Instruction Format

op a, b, c

Syntax Example

SBCS<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Cleared
V	•	= Set if the result is saturated

Description

Subtract source operand 2 (c) from source operand 1 (b), and also subtract the state of the carry flag STATUS32.C. Store the saturated result in the destination register (a). If the carry flag is set after the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used. If the subtraction overflows, the result is saturated to the maximum positive value 0x7FFF_FFFF or the minimum negative value 0x8000_0000. The saturation flag DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix.

Pseudo Code

```
s = sat((sint128) (b - c) - STATUS32.C, 31, &a);    /* SBCS */
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
    STATUS32.C = 0;
}
```

Assembly Code Example

```
SBCS r0,r1,r2      ; Subtract with carry-in r1 and r2 and write the
                   ; saturated result into r0
```

Syntax and Encoding

Instruction Code

SBCS<.f>	a,b,c	00101 bbb 00 100111 F BBB CCCCC AAAAAA
SBCS<.f>	a,b,u6	00101 bbb 01 100111 F BBB uuuuuu AAAAAA
SBCS<.f>	b,b,s12	00101 bbb 10 100111 F BBB ssssss SSSSSS
SBCS<.cc><.f>	b,b,c	00101 bbb 11 100111 F BBB CCCCC 0QQQQQ
SBCS<.cc><.f>	b,b,u6	00101 bbb 11 100111 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SETACC

Function

Set the accumulator value.

Extension Group

DSP Accumulator

Operation

```
if (c & 0x1) {
  switch (c.b1) {
    0: accwide = (int) b;
    1: acclo = (int) b;
    2: acchi = (int) b;
    3: acchi = (int) b; acclo = (int) b;
    4: accwide = (fract) b;
    5: acclo = (fract) b;
    6: acchi = (fract) b;
    7: acclo = (fract) b; acchi = (fract) b;
    8: accwide = (unsigned) b;
    9: acclo = (unsigned) b;
    10: acchi = (unsigned) b;
    11: acclo = (unsigned) b; acchi = (unsigned) b;
  }
} else {
  switch (c.b1) {
    0: ACC0_LO = b;
    1: ACC0_HI = b;
    2: ACC0_GLO.GUARD = b;
    3: ACC0_GHI.GUARD = b;
  }
}
```

Instruction Format

op a,b,c

Syntax Example

SETACC a,b, c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Set a value in the accumulator auxiliary registers with optional sign extension. The return register is ignored.

Although the return value is ignored, using the loop-counter (LP_COUNT/r60) or the program counter (PCL/r63) as a destination operand to the SETACC instruction results in an illegal instruction exception.

The SETACC instruction ignores the destination operand, similar to an SR instruction.

Instruction Encoding

Operand c	Description
Bits [0]	0: One of the auxiliary registers is initialized 1: SETACC sign extends the value and the accumulator including the guard bits are initialized
Bits[9:8]	Denotes the accumulator or the accumulator register to be used in the operation 0: initialize wide accumulator with integer 1: low accumulator 2: high accumulator 3: initialize wide accumulator with a fractional value

Pseudo Code

```

if (c & 0x1) {
    // sign extend into accum
    switch (c >> 8 & 0xf) {
        0: ACC0_LO = b; // init wide accum to integer
            ACC0_HI = b >> 32;
            if (DSP_CTRL.GE)
                ACC0_GHI.GUARD = b >> 32;
        1: ACC0_LO = b; // init low accum to integer
            if (DSP_CTRL.GE)
                ACC0_GLO.GUARD = b >> 32;
        2: ACC0_HI = b; // init high accum to integer
            if (DSP_CTRL.GE)
                ACC0_GHI.GUARD = b >> 32;
        3: ACC0_LO = ACC0_HI = b; // init low and high accum to integer
            if (DSP_CTRL.GE)
                ACC0_GLO.GUARD = ACC0_GHI.GUARD = b >> 32;
        4: if (DSP_CTRL.PA)
            ACC0_LO = 0; // init wide accum to fractional, pre-accum shift
mode
            ACC0_HI = b;
        else
            ACC0_LO = b<<31; // init wide accum to fractional, post-accum
shift mode
            ACC0_HI = b>>1;
            if (DSP_CTRL.GE)
                ACC0_GHI.GUARD = b >> 32;
        5: if (DSP_CTRL.PA)
            ACC0_LO = b; // init low accum to fractional, pre-accum shift
mode
        else
            ACC0_LO = b>>1; // init low accum to fractional, post-accum
shift mode
            if (DSP_CTRL.GE)
                ACC0_GLO.GUARD = b >> 32;
        6: if (DSP_CTRL.PA)
            ACC0_HI = b; // init high accum to fractional, pre-accum shift
mode
        else
            ACC0_HI = b>>1; // init high accum to fractional, post-accum
shift mode
            if (DSP_CTRL.GE)
                ACC0_GHI.GUARD = b >> 32;
    }
}

```

```

7: if (DSP_CTRL.PA)
    // init low and high accum to fractional, pre-accum shift mode
    ACC0_LO = ACC0_HI = b;
else
    // init low and high accum to fractional, post-accum shift mode
    ACC0_LO = ACC0_HI = b>>1;
if (DSP_CTRL.GE)
    ACC0_GLO.GUARD = ACC0_GHI.GUARD = b >> 32;
8: ACC0_LO = b; // init wide accum to unsigned integer
   ACC0_HI = 0;
   if (DSP_CTRL.GE)
       ACC0_GHI.GUARD = 0;
9: ACC0_LO = b; // init low accum to unsigned integer
   if (DSP_CTRL.GE)
       ACC0_GLO.GUARD = 0;
10: ACC0_HI = b; // init high accum to unsigned integer
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = 0;
11: ACC0_LO = ACC0_HI = b; // init low and high accum to unsigned
integer
    if (DSP_CTRL.GE)
        ACC0_GLO.GUARD = ACC0_GHI.GUARD = 0;
    default;; // nothing, retain value
}
} else {
    // raw write to regs
    switch (c >> 8 & 0x3) {
0: ACC0_LO = b;
1: ACC0_HI = b;
2: ACC0_GLO.GUARD = b;
3: ACC0_GHI.GUARD = b;
    }
}
}

```

Assembly Code Example

```

SETACC          ; Set value r1 in the accumulator AUX register as per r2
r0,r1,r2        ; no result

```

Syntax and Encoding

Instruction Code		
SETACC	a, b, c	00101 bbb 00 0011011 BBB CCCCC AAAAAA
SETACC	a, b, u6	00101 bbb 01 0011011 BBB uuuuuu AAAAAA
SETACC	b, b, s12	00101 bbb 10 0011011 BBB ssssss SSSSSS
SETACC<.cc>	b, b, c	00101 bbb 11 0011011 BBB CCCCC 0QQQQQ
SETACC<.cc>	b, b, u6	00101 bbb 11 0011011 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SQRT

Function

Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result

Extension Group

DSP_DIVSQRT!= none

Operation

```
if (cc) {
    acchi = sqrt(c);
    acclo = c - sqrt(c)2;
    b = acchi;
}
```

Instruction Format

op b, c

Syntax Example

SQRT<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result. The flags are updated only if the .F suffix is specified.

Pseudo Code

```
if (cc == true) {
    ACC0_HI = sqrt(c);
    ACC0_LO = c - ACC0_HI*ACC0_HI;
    b = ACC0_HI;
    if (F == 1) {
        Z_flag = (b == 0) ? 1 : 0;
    }
}
```

/* SQRT */

Assembly Code Example

```
SQRT r1,r2      ; Unsigned integer square-root of r2
                ; return result in r1
```

Syntax and Encoding

Instruction Code

SQRT<.f>	b, c	00101bbb001011110BBBCCCCC110000
SQRT<.f>	b, u6	00101bbb011011110BBBuuuuuu110000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SQRTF

Function

Compute the square-root of an unsigned Q31 fraction

Extension Group

DSP_DIVSQRT!= none

Operation

```
if (cc) {
    acchi = sqrt(c<<31);
    acclo = (c<<31) - sqrt(c)2;
    b = acchi;
}
```

Instruction Formats

op b, c

Syntax Example

SQRTF<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Compute the square-root of an unsigned Q31 fraction, yielding a Q31 fractional result. The flags are updated only if the .F suffix is specified.

Pseudo Code

```

if (cc == true) {                                     /* SQRTF */
    i = c<<31; // 64b unsigned
    ACC0_HI = sqrt(i);
    ACC0_LO = (i) - ACC0_HI*ACC0_HI;
    b = ACC0_HI;
    if (F == 1) {
        Z_flag = (b == 0) ? 1 : 0;
    }
}

```

Assembly Code Example

```

SQRTF r1,r2      ; Fractional square-root of r2
                  ; return result in r1

```

Syntax and Encoding

Instruction Code

SQRTF<.f>	b, c	00101bbb001011110BBBCCCCC110001
SQRTF<.f>	b, u6	00101bbb011011110BBBuuuuuu110001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SUBS

Function

Signed 32-bit subtraction; the result is saturated

Extension Group

DSP basic arithmetic

Operation

$$a = \text{SAT32}(b - c)$$

Instruction Format

op a, b, c

Syntax Example

SUBS<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Cleared
V	•	= Set if result is saturated

Description

Obtain the saturating difference of long word operands b and c, and place the result in the destination register a. If the subtraction overflows, the result is saturated to the maximum positive value 0x7fff_ffff or the minimum negative value 0x8000_0000. The saturation flag DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix.

If the set flags suffix (.F) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
s = sat((sint128) b - c, 31, &a);          /* SUBS */
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
    STATUS32.C = 0;
}
```

Assembly Code Example

```
SUBS r0,r1,r2    ; Subtract r2 from r1 and write saturated result into r1
```

Syntax and Encoding

Instruction Code

SUBS<.f>	a,b,c	00101 bbb 00 000111 FBBBCCCCC AAAAAA
SUBS<.f>	a,b,u6	00101 bbb 01 000111 FBBBuuuuuu AAAAAA
SUBS<.f>	b,b,s12	00101 bbb 10 000111 FBBBssssss SSSSSS
SUBS<.cc><.f>	b,b,c	00101 bbb 11 000111 FBBBCCCCC 0QQQQQ
SUBS<.cc><.f>	b,b,u6	00101 bbb 11 000111 FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VABS2H

Function

Absolute value of a two-way 16-bit vector.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = ABS(c.h1);
b.h0 = ABS(c.b0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VABS2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Obtain the pair-wise absolute value of a two-way vector operand, *c*, and place the result in the destination register, *b*. This instruction does not modify any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
l = (c & 0x0000_ffff); /* VABS2H*/
h = ((c>>16) & 0x0000_ffff);
l = ((l & 0x8000) != 0) ? -l : l;
h = ((h & 0x8000) != 0) ? -h : h;
b = (h << 16) | (l & 0x0000_ffff);
```

Assembly Code Example

```
VABS2H r1,r2      ; Absolute value of a two-way 16-bit vector (r2) is
                  ; stored in r1.
```

Syntax and Encoding

Instruction Code		
VABS2H	b, c	00101bbb001011110BBBCCCCC101000
VABS2H	b, limm	00101bbb001011110BBB111110101000
VABS2H	0, c	00101110001011110111CCCCC101000
VABS2H	0, limm	00101110001011110111111110101000
VABS2H	b, u6	00101bbb011011110BBBuuuuuu101000
VABS2H	0, u6	00101110011011110111uuuuuu101000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VABSS2H

Function

Saturated absolute value of a two-way 16-bit vector.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = SAT16 (ABS (c.h1)) ;
b.h0 = SAT16 (ABS (c.b0)) ;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VABSS2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Obtain the pair-wise absolute value of a two-way vector operand, *c*, and place the saturated result in the destination register *b*. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not modify any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
l = (c & 0x0000_ffff); /* VABSS2H*/
h = ((c>>16) & 0x0000_ffff);
s = (l == 0x8000) || (h == 0x8000);
l = l == 0x8000 ? 0x7fff : (((l & 0x8000) != 0) ? -1 : 1);
h = h == 0x8000 ? 0x7fff : (((h & 0x8000) != 0) ? -h : h);
b = (h << 16) | (l & 0x0000_ffff);
DSP_CTRL.SAT |= s
```

Assembly Code Example

```
VABSS2H r1,r2      ; Saturated absolute value of a two-way 16-bit
                   vector (r2) is ;stored in r1.
```

Syntax and Encoding

Instruction Code

VABSS2H	b, c	00101 bbb 00 101111 0 BBB CCCCC 101001
VABSS2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101001
VABSS2H	0, c	00101 110 00 101111 0 111 CCCCC 101001
VABSS2H	0, limm	00101 110 00 101111 0 111 111110 101001
VABSS2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101001
VABSS2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADD2H

Function

Dual 16-bit SIMD addition

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 + c.h1;
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about the vector operands, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADD2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit `b` and `c` operands specify vectors containing two 16-bit elements. Each pair of elements from `b` and `c` is added to form two 16-bit sums. These are assigned to the destination register, if defined.

This instruction does not modify any flags.

An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                /* VADD2H*/
bh = (b >> 16) & 0x0000_ffff;
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
a1 = b1 + c1;
ah = bh + ch;
a = (a1 & 0x0000_ffff) | (ah << 16);

```

Assembly Code Example

```
VADD2H r1,r2,r3    ; dual SIMD 16-bit addition of r2 and r3
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

VADD2H	a, b, c	00101bbb000101000BBBCCCCCAAAAAA
VADD2H	a, b, u6	00101bbb010101000BBBuuuuuuAAAAAA
VADD2H	b, b, s12	00101bbb100101000BBBssssssSSSSSS
VADD2H<.cc>	b, b, c	00101bbb110101000BBBCCCCC0QQQQQ
VADD2H<.cc>	b, b, u6	00101bbb110101000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADD4B

Function

Four-way 8-bit vector addition.

Extension Group

DSP vector arithmetic

Operation

$$\begin{aligned} a.b3 &= b.b3 + c.b3; \\ a.b2 &= b.b2 + c.b2; \\ a.b1 &= b.b1 + c.b1; \\ a.b0 &= b.b0 + c.b0; \end{aligned}$$


Note

$h0$ and $h1$ represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VADD4B a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Pair-wise byte addition of the b and c operands. The results are stored in the a operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

b1l = b & 0x0000_00ff;
b1h = b & 0x0000_ff00;
bhl = b & 0x00ff_0000;
bhh = b & 0xff00_0000;
c1l = c & 0x0000_00ff;
c1h = c & 0x0000_ff00;
chl = c & 0x00ff_0000;
chh = c & 0xff00_0000;
all = b1l + c1l;
alh = b1h + c1h;
ahl = bhl + chl;
ahh = bhh + chh;
a = (all & 0x0000_00ff) | (alh & 0x0000_ff00) | (ahl & 0x00ff_0000) |
    ahh;

```

Assembly Code Example

```

VADD4B r0,r1,r2      ; Add two vectors r1 and r2, and store the result in
                     ; r0;

```

Syntax and Encoding

Instruction Code

VADD4B	a, b, c	00101 bbb 00 100100 0 BBB CCCCC AAAAAA
VADD4B	a, b, u6	00101 bbb 01 100100 0 BBB uuuuuu AAAAAA
VADD4B	b, b, s12	00101 bbb 10 100100 0 BBB ssssss SSSSSS
VADD4B<.cc>	b, b, c	00101 bbb 11 100100 0 BBB CCCCC 0QQQQQ
VADD4B<.cc>	b, b, u6	00101 bbb 11 100100 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUB2H

Function

Dual 16-bit SIMD add and subtract.

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit *b* and *c* operands specify vectors containing two 16-bit elements. The higher 16 bits from *b* and *c* is added to form a 16-bit sum. The lower 16 bits are subtracted to form a 16-bit difference. The sum and difference are assigned to the destination register, if defined.

This instruction does not modify any flags. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if(cc) {                                     /* VADDSUB2H*/
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```

Assembly Code Example

```
VADDSUB2H r1,r2,r3      ; dual SIMD 16-bit addition and subtraction
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VADDSUB2H	a, b, c	00101bbb000101100BBBCCCCCAAAAAA
VADDSUB2H	a, b, u6	00101bbb010101100BBBuuuuuuAAAAAA
VADDSUB2H	b, b, s12	00101bbb100101100BBBsssssssSSSSSS
VADDSUB2H<.cc>	b, b, c	00101bbb110101100BBBCCCCC0QQQQQ
VADDSUB2H<.cc>	b, b, u6	00101bbb110101100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDS2H

Function

Two-way 16-bit vector addition.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = SAT16 (b.h1+c.h1) ;
a.h0 = SAT16 (b.h0+c.h0) ;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VADDS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Pair-wise 16-bit addition of the *b* and *c* operands. The saturated results are stored in the *a* operand. If any of the addition results saturate, the saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

bl = b & 0x0000_ffff;                                     /* VADDS2H/*
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
sl = sat((sint128)bl + cl, 15, al);
sh = sat((sint128)bh + ch, 15, ah);
a = (al & 0x0000_ffff) | (ah << 16);
DSP_CTRL.SAT |= sl || sh;

```

Assembly Code Example

```

VADDS2H r0,r1,r2      ; Add two vectors r1 and r2, and store the saturated
                      ; result in r0

```

Syntax and Encoding

Instruction Code

VADDS2H	a,b,c	00101 bbb 00 010100 1 BBB CCCCC AAAAAA
VADDS2H	a,b,u6	00101 bbb 01 010100 1 BBB uuuuuu AAAAAA
VADDS2H	b,b,s12	00101 bbb 10 010100 1 BBB ssssss SSSSSS
VADDS2H<.cc>	b,b,c	00101 bbb 11 010100 1 BBB CCCCC 0QQQQQ
VADDS2H<.cc>	b,b,u6	00101 bbb 11 010100 1 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUB2H

Function

Dual 16-bit SIMD add and subtract.

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit *b* and *c* operands specify vectors containing two 16-bit elements. The higher 16 bits from *b* and *c* is added to form a 16-bit sum. The lower 16 bits are subtracted to form a 16-bit difference. The sum and difference are assigned to the destination register, if defined.

This instruction does not modify any flags.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                     /* VADDSUB2H*/
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
a1 = b1 + cl;
ah = bh - ch;
a = (a1 & 0x0000_ffff) | (ah << 16);

```

Assembly Code Example

```
VADDSUB2H r1,r2,r3      ; dual SIMD 16-bit addition and subtraction
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

VADDSUB2H	a, b, c	00101 bbb 00 010110 0BBBCCCCC AAAAAA
VADDSUB2H	a, b, u6	00101 bbb 01 010110 0BBBuuuuuu AAAAAA
VADDSUB2H	b, b, s12	00101 bbb 10 010110 0BBBssssss SSSSSS
VADDSUB2H<.cc>	b, b, c	00101 bbb 11 010110 0BBBCCCCC 0QQQQQ
VADDSUB2H<.cc>	b, b, u6	00101 bbb 11 010110 0BBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUBS2H

Function

Two-way 16-bit vector addition and subtraction. The results are saturated and stored.

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = SAT16 (b.h0 + c.h0);
    a.h1 = SAT16 (b.h1 - c.h1);
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUBS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit `b` and `c` operands specify vectors containing two 16-bit elements. The higher 16 bits from `b` and `c` is added to form a 16-bit sum. The lower 16 bits are subtracted to form a 16-bit difference. The saturated results are assigned to the lower and upper half-words of the destination register.

The saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix if the results of this instruction saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                     /* VADDSUBS2H*/
bh = (b >> 16) & 0x0000_ffff;
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)b1 + c1, 15, a1);
sh = sat((sint128)bh - ch, 15, ah);
a = (a1 & 0x0000_ffff) | (ah << 16);
DSP_CTRL.SAT |= s

```

Assembly Code Example

```

VADDSUBS2H r1,r2,r3    ; dual SIMD 16-bit addition and subtraction and
                       ;store the saturated result

```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

VADDSUBS2H	a, b, c	00101bbb000101101BBBCCCCCAAAAAA
VADDSUBS2H	a, b, u6	00101bbb010101101BBBuuuuuuAAAAAA
VADDSUBS2H	b, b, s12	00101bbb100101101BBBssssssSSSSSS
VADDSUBS2H<.cc>	b, b, c	00101bbb110101101BBBCCCCC0QQQQQ
VADDSUBS2H<.cc>	b, b, u6	00101bbb110101101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VALGN2H

Function

Compose a dual 16-bit vector from the least-significant 16 bits of the *b* operand and the most-significant 16 bits of the *c* operand.

Extension Group

DSP vector unpacking

Operation

```
{
  a.h1 = b.h0;
  a.h0 = c.h1;
}
```



Note

h0 and *h1* represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VALGN2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector from the least-significant 16 bits of the *b* operand and the most-significant 16 bits of the *c* operand. This instruction is used for realigning vector data before load and store operations. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
a.h1 = b.h0;
a.h0 = c.h1;                                     /* VALGN2H*/
```

Assembly Code Example

```
VALGN2H r0,r1,r2      ; compose a vector from lower 16-bits of r1 and
                      ;higher 16-bits of r2
```

Syntax and Encoding

		Instruction Code
VALGN2H	a,b,c	00101bbb000011010BBBCCCCCAAAAAA
VALGN2H	a,b,u6	00101bbb010011010BBBuuuuuuAAAAAA
VALGN2H	b,b,s12	00101bbb100011010BBBssssssSSSSSS
VALGN2H<.cc>	b,b,c	00101bbb110011010BBBCCCCC0QQQQQ
VALGN2H<.cc>	b,b,u6	00101bbb110011010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VASL2H

Function

Perform a two-way arithmetic shift left operation 16-bit vectors. The shift amount is specified in the second operand.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = b.h1 << c.h1;
a.h0 = b.h0 << c.h0;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASL2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Left shift the 16-bit vector elements of the operand *b* by the two-way vector value in operand *c*. The shift amount is unsigned and only the bottom four bits of the vector shift amount are used. This instruction never set flags. The resulting vector is stored in the destination operand. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

bl = b & 0x0000_ffff;
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_000f;
ch = (c >> 16) & 0x0000_000f;
al = bl << cl;
ah = bh << ch;
a = (al & 0x0000_ffff) | (ah << 16);
/* VASL2H/ */

```

Assembly Code Example

```

VASL2H r0,r1,r2      ; Shift the operand r1 left by scalar r2, and return
                     ; the result to r0.

```

Syntax and Encoding

Instruction Code

VASL2H	a,b,c	00101bbb001000010BBBCCCCCAAAAAA
VASL2H	a,b,u6	00101bbb011000010BBBuuuuuuAAAAAA
VASL2H	b,b,s12	00101bbb101000010BBBssssssSSSSSS
VASL2H<.cc>	b,b,c	00101bbb111000010BBBCCCCC0QQQQQ
VASL2H<.cc>	b,b,u6	00101bbb111000010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VASLS2H

Function

Two-way 16-bit vector arithmetic shift left and store the saturated result.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = SAT16 (b.h1 << c.h1);
a.h0 = SAT16 (b.h0 << c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASLS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Left shift the 16-bit vector elements of the operand *b* by the two-way vectors in the *c* operand. The shift amount can be negative, effectively shifting right. The resulting vector is saturated and stored in the destination operand. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

If the shift amount is smaller than -15, the shift amount saturates to -15.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                     /* VASLS2H/*
bh = (b >> 16) & 0x0000_ffff;
cl = sext16(c & 0x0000_ffff);
ch = c >> 16;
cl = cl > -15 ? cl : -15;
ch = ch > -15 ? ch : -15;
if (cl >= 0)
    sl = sat((sint128)b1 << cl, 15, &a1);
else
    al = b1 >> -cl;
if (ch >= 0)
    sh = sat((sint128)bh << ch, 15, &ah);
else
    ah = bh >> -ch;
DSP_CTRL.SAT |= sl || sh
a = (al & 0x0000_ffff) | (ah << 16);

```

Assembly Code Example

```

VASLS2H r0,r1,r2      ; Shift the operand r1 left by scalar r2, and return
                       ;the saturated result to r0.

```

Syntax and Encoding

Instruction Code		
VASLS2H	a,b,c	00101bbb001000011BBBCCCCCAAAAAA
VASLS2H	a,b,u6	00101bbb011000011BBBuuuuuuAAAAAA
VASLS2H	b,b,s12	00101bbb101000011BBBssssssSSSSSS
VASLS2H<.cc>	b,b,c	00101bbb111000011BBBCCCCC0QQQQQ
VASLS2H<.cc>	b,b,u6	00101bbb111000011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VASR2H

Function

Perform a two-way arithmetic shift right operation on 16-bit vectors. The shift amount is specified in the second operand.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = b.h1 >> c.h1;
a.h0 = b.h0 >> c.h0;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASR2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Right shift the 16-bit vector elements of the operand *b* by the scalar value in the *c* operand. The shift amount is unsigned and only the lower 4 bits of the vector shift amount is used. The resulting vector is stored in the destination operand. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

bl = b & 0x0000_ffff;
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_000f;
ch = (c >> 16) & 0x0000_000f;
al = bl >> cl;
ah = bh >> ch;
a = (al & 0x0000_ffff) | (ah << 16);

```

/* VASR2H/*

Assembly Code Example

```

VASR2H r0,r1,r2      ; Shift the operand r1 right by scalar r2, and
                     ;return the result to r0.

```

Syntax and Encoding

Instruction Code

VASR2H	a,b,c	00101bbb001000100BBBCCCCCAAAAAA
VASR2H	a,b,u6	00101bbb011000100BBBuuuuuuAAAAAA
VASR2H	b,b,s12	00101bbb101000100BBBssssssSSSSSS
VASR2H<.cc>	b,b,c	00101bbb111000100BBBCCCCC0QQQQQ
VASR2H<.cc>	b,b,u6	00101bbb111000100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VASRS2H

Function

Perform a two-way arithmetic shift right operation on 16-bit vectors. The result is saturated.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = SAT16 (b.h1 >> c.h1);
a.h0 = SAT16 (b.h0 >> c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASRS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Right shift the 16-bit vector elements of the operand *b* by the two-way vectors in the *c* operand. The shift amount can be negative, effectively shifting left. The resulting vector is saturated and stored in the destination operand. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the *.F* suffix. This instruction does not update any STATUS32 flags.

If the shift amount is greater than 15, the shift amount saturates to 15.

Pseudo Code

```

bl = b & 0x0000_ffff;                                     /* VASRS2H/*
bh = (b >> 16) & 0x0000_ffff;
cl = sext16(c & 0x0000_ffff);
ch = c >> 16;
cl = cl < 15 ? cl : 15;
ch = ch < 15 ? ch : 15;
if (cl < 0)
    sl = sat((sint128)bl << -cl, 15, &al);
else
    al = bl >> cl;
if (ch < 0)
    sh = sat((sint128)bh << -ch, 15, &ah);
else
    ah = bh >> ch;
DSP_CTRL.SAT |= sl || sh
a = (al & 0x0000_ffff) | (ah << 16);

```

Assembly Code Example

```

VASRS2H r0,r1,r2      ; Shift the operand r1 right by scalar r2, and
                      ;return the saturated result to r0.

```

Syntax and Encoding

Instruction Code		
VASRS2H	a,b,c	00101bbb001000101BBBCCCCCAAAAAA
VASRS2H	a,b,u6	00101bbb011000101BBBuuuuuuAAAAAA
VASRS2H	b,b,s12	00101bbb101000101BBBssssssSSSSSS
VASRS2H<.cc>	b,b,c	00101bbb111000101BBBCCCCC0QQQQQ
VASRS2H<.cc>	b,b,u6	00101bbb111000101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

VASRSR2H

Function

Perform a two-way arithmetic shift right operation on 16-bit vectors. The result is rounded and saturated.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = SAT16 (RND16 (b.h1 >> c.h1)) ;
a.h0 = SAT16 (RND16 (b.h0 >> c.h0)) ;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASRSR2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Right shift the 16-bit vector elements of the operand *b* by the two-way vector in the *c* operand. Bits shifted out at the LSB side contributes to rounding. Rounding mode is defined in `DSP_CTRL.RM`. If the shift amount is negative, the vectors are shifted left. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the *.F* suffix.

If the shift amount is greater than 15, the shift amount saturates to 15.

This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

bl = b & 0x0000_ffff;                                     /* VASRSR2H/*
bh = (b >> 16) & 0x0000_ffff;
cl = sext16(c & 0x0000_ffff);
ch = c >> 16;
cl = cl < 16 ? cl : 16;
ch = ch < 16 ? ch : 16;
cl = cl < -16 ? -16 : cl;
ch = ch < -16 ? -16 : ch;
if (cl < 0)
    sl = sat((sint128)bl << -cl, 15, &al);
else
    al = round(bl, cl) >> cl;
if (ch < 0)
    sh = sat((sint128)bh << -ch, 15, &ah);
else
    ah = round(bh, ch) >> ch;
DSP_CTRL.SAT |= sl || sh
a = (al & 0x0000_ffff) | (ah << 16);

```

Assembly Code Example

```

VASRSR2H r0,r1,r2    ; Shift the operand r1 right by scalar r2, and
                    ;return the saturated and rounded result to r0.

```

Syntax and Encoding

Instruction Code

VASRSR2H	a,b,c	00101bbb001000111BBBCCCCCAAAAAA
VASRSR2H	a,b,u6	00101bbb011000111BBBuuuuuuAAAAAA
VASRSR2H	b,b,s12	00101bbb101000111BBBssssssSSSSSS
VASRSR2H<.cc>	b,b,c	00101bbb111000111BBBCCCCC0QQQQQ
VASRSR2H<.cc>	b,b,u6	00101bbb111000111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VEXT2BHL

Function

Compose a dual 16-bit vector by zero-extending lower two bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = 0;
b.b2 = c.b1;
b.b1 = 0;
b.b0 = c.b0;
```



Note

b0, b1, b2, and b3 represent the 8-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VEXT2BHL b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by zero-extending the two lower bytes of the *c* operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = (c & 0x0000_00ff) | ((c & 0x0000_ff00) << 8); /* VEXT2BHL*/
```

Assembly Code Example

```
VEXT2BHL r1,r2      ; zero extend two lower bytes from r2 into 16-bit
                    ; words in r1
```

Syntax and Encoding

Instruction Code		
VEXT2BHL	b, c	00101 bbb 00 101111 0 BBB CCCCC 100100
VEXT2BHL	b, limm	00101 bbb 00 101111 0 BBB 111110 100100
VEXT2BHL	0, c	00101 110 00 101111 0 111 CCCCC 100100
VEXT2BHL	0, limm	00101 110 00 101111 0 111 111110 100100
VEXT2BHL	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100100
VEXT2BHL	0, u6	00101 110 01 101111 0 111 uuuuuu 100100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VEXT2BHLF

Function

Compose a vector with two Q15 elements by expanding the lower two Q7 bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = c.b1;
b.b2 = 0;
b.b1 = c.b0;
b.b0 = 0;
```



Note

b0, b1, b2, and b3 represent the Q7 elements of an operand.

Instruction Format

op b,c

Syntax Example

VEXT2BHLF b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

MSB-aligned expansion of the two lower bytes (Q7) of an operand into two Q15 elements. This instruction does not update any flags.

Pseudo Code

```
b = ((c & 0x0000_00ff)<<8) | ((c & 0x0000_ff00)<<16); /* VEXT2BHLF*/
```

Assembly Code Example

```
VEXT2BHLF r0,r1      ; Expand and pack 2 lower bytes from r1 into
                      ; 2 16b words in r0
```

Syntax and Encoding

Instruction Code

VEXT2BHLF	b, c	00101 bbb 00 101111 0 BBB CCCCC 100000
VEXT2BHLF	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VEXT2BHM

Function

Compose a dual 16-bit vector by zero-extending the higher two bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = 0;
b.b2 = c.b3;
b.b1 = 0;
b.b0 = c.b2;
```



Note

b0, b1, b2, and b3 represent the 8-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VEXT2BHM b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by zero-extending the higher two bytes of an operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = ((c >> 16) & 0x0000_00ff) | ((c >> 8) & 0x00ff_0000); /* VEXT2BHM*/
```

Assembly Code Example

```
VEXT2BHM r1,r2      ; zero extend two higher bytes from r2 into 16-bit
                    ; words in r1
```

Syntax and Encoding

		Instruction Code
VEXT2BHM	b, c	00101 bbb 00 101111 0 BBB CCCCC 100101
VEXT2BHM	b, limm	00101 bbb 00 101111 0 BBB 111110 100101
VEXT2BHM	0, c	00101 110 00 101111 0 111 CCCCC 100101
VEXT2BHM	0, limm	00101 110 00 101111 0 111 111110 100101
VEXT2BHM	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100101
VEXT2BHM	0, u6	00101 110 01 101111 0 111 uuuuuu 100101

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VEXT2BHMF

Function

Compose a vector with two Q15 elements created by expanding the higher two Q7 bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = c.b3;
b.b2 = 0;
b.b1 = c.b2;
b.b0 = 0;
```



Note

b0, b1, b2, and b3 represent the Q7 elements of an operand.

Instruction Format

op b,c

Syntax Example

VEXT2BHMF b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

MSB_aligned expansion of the higher two Q7 bytes of an operand into two Q15 elements This instruction does not modify any flags.

Pseudo Code

```
b = ((c & 0x00ff_0000)>>8) | ((c & 0xff00_0000)); /* VEXT2BHMF*/
```

Assembly Code Example

```
VEXT2BHMF r1,r2      ; Expand and pack 2 lower bytes from r1
                      ; into 2 16b words in r0
```

Syntax and Encoding

Instruction Code

VEXT2BHMF	b, c	00101 bbb 00 101111 0 BBB CCCCC 100001
VEXT2BHMF	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VLSR2H

Function

Perform a two-way logical shift right operation on 16-bit vectors.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = unsigned (b.h1) >> c.h1;
a.h0 = unsigned (b.h0) >> c.h0;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VLSR2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Logic shift right the 16-bit vector elements of the operand *b* by the scalar *c*. The resulting vector is stored in the destination operand. The shift amount is unsigned and only the lower 4 bits of the vector shift amount is used. This instruction does not update any STATUS32 flags.

Pseudo Code

```

bl = b & 0x0000_ffff;
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_000f;
ch = (c >> 16) & 0x0000_000f;
al = bl >> cl;
ah = bh >> ch;
a = (al & 0x0000_ffff) | (ah << 16);
/* VLSR2H/ */

```

Assembly Code Example

```

VLSR2H r0,r1,r2      ; Logic shift right the operand r1 by scalar r2, and
                    ;return the result to r0.

```

Syntax and Encoding

Instruction Code

VLSR2H	a, b, c	00101bbb001000110BBBCCCCCAAAAAA
VLSR2H	a, b, u6	00101bbb011000110BBBuuuuuuAAAAAA
VLSR2H	b, b, s12	00101bbb101000110BBBssssssSSSSSS
VLSR2H<.cc>	b, b, c	00101bbb111000110BBBCCCCC0QQQQQ
VLSR2H<.cc>	b, b, u6	00101bbb111000110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2H

Function

Signed multiplication and accumulation of two 16-bit vectors.

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
A.w1 = acchi+(b.h1 * c.h1);
A.w0 = acclo+(b.h0 * c.h0);
```



Note

`h0` and `h1` represent the 16-bit elements of a 32-bit operand.

Instruction Format

op A,b,c

Syntax Example

VMAC2H A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed integer multiplication of the higher 16 bits of `b` and `c` and add the product to the high accumulator. Similarly, perform a signed integer multiplication of the lower 16 bits of `b` and `c` and add the product to the low accumulator. Return the products as a two-way 32-bit vector in two consecutive cycles in a register pair `A`. `A` should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (`LP_COUNT/r60`) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```

acclo += b.lo * c.lo;                                /* VMAC2H*/
acchi += b.h1 * c.h1;
if (DSP_CTRL.GE)                                     // only if guard bits
are enabled
    ACC0_GLO = acclo >> 32;
    ACC0_GHI = acchi >> 32;
ACC0_LO = acclo & 0xffff_ffff;
ACC0_HI = acchi & 0xffff_ffff;
a = (ACC0_HI<<32) | ACC0_LO; // 64b result

```

Assembly Code Example

```

VMAC2H r0,r2,r3      ;Signed integer multiply two vectors r2 and r3 and
                     ; return result in r0 and r1

```

Syntax and Encoding

Instruction Code		
VMAC2H	a,b,c	00101bbb000111100BBBCCCCCAAAAAA
VMAC2H	a,b,u6	00101bbb010111100BBBuuuuuuAAAAAA
VMAC2H	b,b,s12	00101bbb100111100BBBssssssSSSSSS
VMAC2H<.cc>	b,b,c	00101bbb110111100BBBCCCCC0QQQQQ
VMAC2H<.cc>	b,b,u6	00101bbb110111100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2HF

Function

Two-way signed multiplication and accumulation of two 16-bit fractional vectors. The result is saturated and stored.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16((acchi+(b.h1 * c.h1))<<1);
a.h0 = SAT16((acclo+=(b.h0 * c.h0))<<1);
```



Note

`h0` and `h1` represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMAC2HF a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and add the product to the high accumulator to get the result. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c` and add the product to the low accumulator to get the result. Return the results as a two-way 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMAC2HF*/
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        // guard bits are enabled
        acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(acclo, 31, &alo);
        acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(acchi, 31, &ahi);
    else
        // guard bits disabled
        sl = sat((b.h0 * c.h0 << 1), 31, &addlo);
        sh = sat((b.h1 * c.h1 << 1), 31, &addhi);
        acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(acclo, 31, &alo) || sl;
        acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(acchi, 31, &ahi) || sh;
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT| = sl | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMAC2HF r0,r1,r2      ; Signed fractional multiply-accumulate two vectors
                       ;r1 and r2 and return result in r0

```

Syntax and Encoding

Instruction Code		
VMAC2HF	a,b,c	00101bbb000111101BBBCCCCCAAAAAA
VMAC2HF	a,b,u6	00101bbb010111101BBBuuuuuuAAAAAA
VMAC2HF	b,b,s12	00101bbb100111101BBBssssssSSSSSS
VMAC2HF<.cc>	b,b,c	00101bbb110111101BBBCCCCC0QQQQQ
VMAC2HF<.cc>	b,b,u6	00101bbb110111101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2HFR

Function

Two-way signed multiplication and accumulation of two 16-bit fractional vectors. The result is rounded and saturated.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16 (RND16 ((acchi += (b.h1 * c.h1)) << 1));
a.h0 = SAT16 (RND16 ((acclo += (b.h0 * c.h0)) << 1));
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMAC2HFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of *b* and *c* and add the product to the high accumulator to get the result. Similarly, perform a signed fractional multiplication of the lower 16 bits of *b* and *c* and add the product to the low accumulator to get the result. Return the results as a two-way rounded 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMAC2HFR*/
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        sl = sat((b.h0 * c.h0 << 1), 31, &addlo);
        sh = sat((b.h1 * c.h1 << 1), 31, &addhi);
        acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo) || sl;
        acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi) || sh;
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMAC2HFR r0,r1,r2      ; Signed fractional multiply two vectors r1 and r2
                        ;and return result in r0 with saturation and rounding

```

Syntax and Encoding

Instruction Code		
VMAC2HFR	a,b,c	00101bbb000111111BBBCCCCCAAAAAA
VMAC2HFR	a,b,u6	00101bbb010111111BBBuuuuuuAAAAAA
VMAC2HFR	b,b,s12	00101bbb100111111BBBssssssSSSSSS
VMAC2HFR<.cc>	b,b,c	00101bbb110111111BBBCCCCC0QQQQQ
VMAC2HFR<.cc>	b,b,u6	00101bbb110111111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2HNFR

Function

Two-way signed multiplication and accumulation of two 16-bit fractional vectors. The result is rounded and saturated, and returned without a fractional shift.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16(RND16(acchi+(b.h1 * c.h1)));
a.h0 = SAT16(RND16(acclo+(b.h0 * c.h0)));
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMAC2HNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of *b* and *c* and add the product to the high accumulator to get the result. Similarly, perform a signed fractional multiplication of the lower 16 bits of *b* and *c* and add the product to the low accumulator to get the result. Return the results as a two-way rounded 16-bit fractional vector. The result is not shifted by one bit. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMAC2HNFR*/
addlo = b.h0 * c.h0;
addhi = b.h1 * c.h1;
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= sl | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMAC2HNFR r0,r1,r2      ; Signed fractional multiply two vectors r1 and r2
                        ;and return result in r0 with saturation and rounding

```

Syntax and Encoding

Instruction Code

VMAC2HNFR	a,b,c	00110 bbb 00 010001 0 BBB CCCCC AAAAAA
VMAC2HNFR	a,b,u6	00110 bbb 01 010001 0 BBB uuuuuu AAAAAA

VMAC2HNFR	b, b, s12	00110 bbb 10 0100010 BBB ssssss SSSSSS
VMAC2HNFR<.cc>	b, b, c	00110 bbb 11 0100010 BBB CCCCC 0QQQQQ
VMAC2HNFR<.cc>	b, b, u6	00110 bbb 11 0100010 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2HU

Function

Unsigned multiplication and accumulation of two 16-bit vectors.

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
A.w1 = acchi+(b.h1 * c.h1);
A.w0 = acclo+(b.h0 * c.h0);
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op A,b,c

Syntax Example

VMAC2HU A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an unsigned integer multiplication of the higher 16 bits of *b* and *c* and add the product to the high accumulator. Similarly, perform an unsigned integer multiplication of the lower 16 bits of *b* and *c* and add the product to the low accumulator. Return the products as a two-way 32-bit vector in two consecutive cycles in a register pair, *A*. *A* should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```

acclo += b.h0 * c.h0;                                /* VMAC2HU*/
acchi += b.h1 * c.h1;
if (DSP_CTRL.GE)                                     // only if guard bits
are enabled
    ACC0_GLO = acclo >> 32;
    ACC0_GHI = acchi >> 32;
ACC0_LO = acclo & 0xffff_ffff;
ACC0_HI = acchi & 0xffff_ffff;
a = (ACC0_HI<<32) | ACC0_LO; // 64b result

```

Assembly Code Example

```

VMAC2HU r0,r2,r3      ;Unsigned integer multiply two vectors r2 and r3 and
                      ; return result in r0 and r1

```

Syntax and Encoding

Instruction Code		
VMAC2HU	a,b,c	00101bbb000111110BBBCCCCCAAAAAA
VMAC2HU	a,b,u6	00101bbb010111110BBBuuuuuuAAAAAA
VMAC2HU	b,b,s12	00101bbb100111110BBBssssssSSSSSS
VMAC2HU<.cc>	b,b,c	00101bbb110111110BBBCCCCC0QQQQQ
VMAC2HU<.cc>	b,b,u6	00101bbb110111110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAX2H

Function

Compare two-way 16-bit vectors and return the maximum.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = max (b.h1, c.h1);
a.h0 = max (b.h0, c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VMAX2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compare the 16-bit signed vector elements of the operand `b` and `c`. The maximum value of the two elements is stored in the destination operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

bl = (b & 0x0000_ffff) << 16;                                     /* VMAX2H/*
bh = ((b >> 16) & 0x0000_ffff) << 16;
cl = (c & 0x0000_ffff) << 16;
ch = ((c >> 16) & 0x0000_ffff) << 16;
al = bl >= cl ? bl : cl;
ah = bh >= ch ? bh : ch;
a = ((al >> 16) & 0x0000_ffff) | ah;

```

Assembly Code Example

```

VMAX2H r0,r1,r2           ; Compare the 16-bit vector elements of the operands
                           ;r2 and r1, and return the maximum to r0.

```

Syntax and Encoding

Instruction Code

VMAX2H	a,b,c	00101bbb001001001BBBCCCCCAAAAAA
VMAX2H	a,b,u6	00101bbb011001001BBBuuuuuuAAAAAA
VMAX2H	b,b,s12	00101bbb101001001BBBssssssSSSSSS
VMAX2H<.cc>	b,b,c	00101bbb111001001BBBCCCCC0QQQQQ
VMAX2H<.cc>	b,b,u6	00101bbb111001001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMIN2H

Function

Compare two-way 16-bit signed vectors and return the minimum.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = min (b.h1, c.h1);
a.h0 = min (b.h0, c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VMIN2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compare the 16-bit signed vector elements of the operand **b** and **c**. The minimum value of the two elements is stored in the destination operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

bl = (b & 0x0000_ffff) << 16;                                     /* VMIN2H/*
bh = ((b >> 16) & 0x0000_ffff) << 16;
cl = (c & 0x0000_ffff) << 16;
ch = ((c >> 16) & 0x0000_ffff) << 16;
al = bl < cl ? bl : cl;
ah = bh < ch ? bh : ch;
a = ((al >> 16) & 0x0000_ffff) | ah;

```

Assembly Code Example

```

VMIN2H r0,r1,r2          ; Compare the 16-bit vector elements of the operands
                          ;r2 and r1, and return the minimum to r0.

```

Syntax and Encoding

Instruction Code

VMIN2H	a,b,c	00101bbb001001011BBBCCCCCAAAAAA
VMIN2H	a,b,u6	00101bbb011001011BBBuuuuuuAAAAAA
VMIN2H	b,b,s12	00101bbb101001011BBBssssssSSSSSS
VMIN2H<.cc>	b,b,c	00101bbb111001011BBBCCCCC0QQQQQ
VMIN2H<.cc>	b,b,u6	00101bbb111001011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2H

Function

Dual 16-bit SIMD multiplication.

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
if (cc) {
    acc.w0 = A.w0 = b.h0 * c.h0;
    acc.w1 = A.w1 = b.h1 * c.h1;
}
```

Instruction Format

op A, b, c

Syntax Example

VMPY2H A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed integer multiplication of the higher 16 bits of *b* and *c* and store the product to the high accumulator. Similarly, perform a signed integer multiplication of the lower 16 bits of *b* and *c* and store the product to the low accumulator. Return the products as a two-way 32-bit vector in two consecutive cycles in a register pair *A*. *A* should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```

acclo = b.h0 * c.h0;                                     /* VMPY2H*/
acchi = b.h1 * c.h1;
if (DSP_CTRL.GE)                                         // only if guard bits
are enabled
    ACC0_GLO = acclo >> 32;
    ACC0_GHI = acchi >> 32;
ACC0_LO = acclo & 0xffff_ffff;
ACC0_HI = acchi & 0xffff_ffff;
a = (ACC0_HI<<32) | ACC0_LO; // 64b result

```

Assembly Code Example

```
VMPY2H r0,r2,r3 ; dual SIMD 16-bit multiplication
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

Instruction Code

VMPY2H	a, b, c	00101bbb0001111000BBBCCCCCAAAAAA
VMPY2H	a, b, u6	00101bbb0101111000BBBuuuuuuAAAAAA
VMPY2H	b, b, s12	00101bbb1001111000BBBssssssSSSSSS
VMPY2H<.cc>	b, b, c	00101bbb1101111000BBBCCCCC0QQQQQ
VMPY2H<.cc>	b, b, u6	00101bbb1101111000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2HF

Function

Two-way signed multiplication of two 16-bit fractional vectors. The result is saturated and stored.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16((acchi = (b.h1 * c.h1)) << 1);
a.h0 = SAT16((acclo = (b.h0 * c.h0)) << 1);
```



Note

`h0` and `h1` represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMPY2HF a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and store the product to the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c` and store the product to the low accumulator. Return the products as a two-way 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMPY2HF*/
if (DSP_CTRL.PA)
// pre-accumulate fractional shift mode
if (DSP_CTRL.GE)
// guard bits are enabled
addlo = b.h0 * c.h0 << 1;
addhi = b.h1 * c.h1 << 1;
acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
ACC0_LO         = acclo & 0xffff_ffff;
s1              = sat(acclo, 31, &alo);
acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
ACC0_HI         = acchi & 0xffff_ffff;
sh              = sat(acchi, 31, &ahi);
else
// guard bits disabled
s1 = sat((b.h0 * c.h0 << 1), 31, &addlo);
sh = sat((b.h1 * c.h1 << 1), 31, &addhi);
acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
ACC0_GLO.G      = 0;
ACC0_LO         = acclo & 0xffff_ffff;
s1              = sat(acclo, 31, &alo) || s1;
acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.G      = 0;
ACC0_HI         = acchi & 0xffff_ffff;
sh              = sat(acchi, 31, &ahi) || sh;
else
// post-accumulate fractional shift mode
addlo = b.h0 * c.h0;
addhi = b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMPY2HF r0,r1,r2      ; Signed fractional multiply two vectors r1 and r2
                      ;and return result in r0

```

Syntax and Encoding

Instruction Code		
VMPY2HF	a,b,c	00101bbb000111001BBBCCCCCAAAAAA
VMPY2HF	a,b,u6	00101bbb010111001BBBuuuuuuAAAAAA
VMPY2HF	b,b,s12	00101bbb100111001BBBssssssSSSSSS
VMPY2HF<.cc>	b,b,c	00101bbb110111001BBBCCCCC0QQQQQ
VMPY2HF<.cc>	b,b,limm	00101bbb110111001BBB1111100QQQQQ

VMPY2HF<.cc> b,b,u6 00101**bbb**11**011100**1**BBB**uuuuuu**1QQQQQ**

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2HFR

Function

Two-way signed multiplication of two 16-bit fractional vectors. The result is saturated, rounded, and stored.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16(RND16((acchi = (b.h1 * c.h1)) << 1));
a.h0 = SAT16(RND16((acclo = (b.h0 * c.h0)) << 1));
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMPY2HFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of *b* and *c* and store the product to the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of *b* and *c* and store the product to the low accumulator. Return the products as a two-way rounded 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMPY2HFR*/
if (DSP_CTRL.PA)
// pre-accumulate fractional shift mode
if (DSP_CTRL.GE)
// guard bits are enabled
addlo = b.h0 * c.h0 << 1;
addhi = b.h1 * c.h1 << 1;
acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
ACC0_LO         = acclo & 0xffff_ffff;
sl              = sat(round(acclo, 16), 31, &alo);
acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
ACC0_HI         = acchi & 0xffff_ffff;
sh              = sat(round(acchi, 16), 31, &ahi);
else
// guard bits disabled
sl = sat((b.h0 * c.h0 << 1), 31, &addlo);
sh = sat((b.h1 * c.h1 << 1), 31, &addhi);
acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
ACC0_GLO.G      = 0;
ACC0_LO         = acclo & 0xffff_ffff;
sl              = sat(round(acclo, 16), 31, &alo) || sl;
acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.G      = 0;
ACC0_HI         = acchi & 0xffff_ffff;
sh              = sat(round(acchi, 16), 31, &ahi) || sh;
else
// post-accumulate fractional shift mode
addlo = b.h0 * c.h0;
addhi = b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMPY2HFR r0,r1,r2      ; Signed fractional multiply two vectors r1 and r2
                        ;and return the saturated and rounded result in r0

```

Syntax and Encoding

Instruction Code		
VMPY2HFR	a,b,c	00101bbb000111011BBBCCCCCAAAAAA
VMPY2HFR	a,b,u6	00101bbb010111011BBBuuuuuuAAAAAA
VMPY2HFR	b,b,s12	00101bbb100111011BBBssssssSSSSSS
VMPY2HFR<.cc>	b,b,c	00101bbb110111011BBBCCCCC0QQQQQ
VMPY2HFR<.cc>	b,b,u6	00101bbb110111011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2HU

Function

Dual unsigned 16-bit SIMD multiplication.

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
if (cc) {
    acc.w0 = A.w0 = b.h0 * c.h0;
    acc.w1 = A.w1 = b.h1 * c.h1;
}
```

Instruction Format

op A,b, c

Syntax Example

VMPY2HU A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an unsigned integer multiplication of the higher 16 bits of *b* and *c* and store the product to the high accumulator. Similarly, perform an unsigned integer multiplication of the lower 16 bits of *b* and *c* and store the product to the low accumulator. Return the products as a two-way 32-bit vector in two consecutive cycles in a register pair *A*. *A* should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```

acclo = b.h0 * c.h0;                                     /* VMPY2HU*/
acchi = b.h1 * c.h1;
if (DSP_CTRL.GE)          // only if guard bits
are enabled
    ACC0_GLO = acclo >> 32;
    ACC0_GHI = acchi >> 32;
ACC0_LO = acclo & 0xffff_ffff;
ACC0_HI = acchi & 0xffff_ffff;
a = (ACC0_HI<<32) | ACC0_LO; // 64b result

```

Assembly Code Example

```

VMPY2HU r0,r2,r3    ; dual SIMD 16-bit unsigned multiplication. The
                    ;result is stored in (r1,r0)

```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

VMPY2HU	a, b, c	00101bbb000111010BBBCCCCCAAAAAA
VMPY2HU	a, b, u6	00101bbb010111010BBBuuuuuuAAAAAA
VMPY2HU	b, b, s12	00101bbb100111010BBBssssssSSSSSS
VMPY2HU<.cc>	b, b, c	00101bbb110111010BBBCCCCC0QQQQQ
VMPY2HU<.cc>	b, b, u6	00101bbb110111010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2HWF

Function

Two-way signed multiplication of two 16-bit fractional vectors. A saturated 64-bit result is returned.

Extension Group

DSP Vector 16x16 MAC

Operation

```
A.w1 = SAT32 ( (acchi = (b.h1 * c.h1)) <<1 );
A.w0 = SAT32 ( (acchi = (b.h0 * c.h0)) <<1 );
```



Note

`h0` and `h1` represent the 16-bit elements of a 32-bit operand.

Instruction Format

op A,b,c

Syntax Example

VMPY2HWF A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and store the product to the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c` and store the product to the low accumulator. Return the products as a two-way 32-bit fractional vector in a register pair `A`. `A` should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (`LP_COUNT/r60`) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit write-back.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMPY2HWF*/
if (DSP_CTRL.PA)
  // pre-accumulate fractional shift mode
  if (DSP_CTRL.GE)
    // guard bits are enabled
    addlo = b.h0 * c.h0 << 1;
    addhi = b.h1 * c.h1 << 1;
    acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo, 31, &alo);
    acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi, 31, &ahi);
  else
    // guard bits disabled
    sl = sat((b.h0 * c.h0 << 1), 31, &addlo);
    sh = sat((b.h1 * c.h1 << 1), 31, &addhi);
    acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl |= sat(acclo, 31, &alo);
    acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh |= sat(acchi, 31, &ahi);
  else
    // post-accumulate fractional shift mode
    addlo = b.h0 * c.h0;
    addhi = b.h1 * c.h1;

```



```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GHI.V | ACC0_GLO.V;
a = (ahi << 32) | (alo & 0xffff_ffff);

```

Assembly Code Example

```

VMPY2HWF r0,r2,r3      ; Signed fractional multiply two vectors r3 and r2
                        ;and return result in r0, r1

```

Syntax and Encoding

Instruction Code		
VMPY2HWF	a,b,c	00101bbb001000000BBBCCCCCAAAAAA
VMPY2HWF	a,b,u6	00101bbb011000000BBBuuuuuuAAAAAA
VMPY2HWF	b,b,s12	00101bbb101000000BBBssssssSSSSSS
VMPY2HWF<.cc>	b,b,c	00101bbb111000000BBBCCCCC0QQQQQ
VMPY2HWF<.cc>	b,b,u6	00101bbb111000000BBBuuuuuu1QQQQQ

VMSUB2HF

Function

Two-way signed multiplication and subtraction of two 16-bit fractional vectors. The result is saturated and stored.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16((acchi -= (b.h1*c.h1)) <<1);
a.h0 = SAT16((acclo -= (b.h0*c.h0)) <<1);
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VMSUB2HF a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and subtract the product from the high accumulator and store the result in the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c`, subtract the product from the low accumulator, and store the result in the low accumulator. Return the results as a two-way 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMSUB2HF*/
if (DSP_CTRL.PA)
  // pre-accumulate fractional shift mode
  if (DSP_CTRL.GE)
    addlo = -b.h0 * c.h0 << 1;
    addhi = -b.h1 * c.h1 << 1;
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1
: ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo, 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1
: ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi, 31, &ahi);
  else
    // guard bits disabled
    sl = sat(-(b.h0 * c.h0 << 1), 31, &addlo);
    sh = sat(-(b.h1 * c.h1 << 1), 31, &addhi);
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo, 31, &alo) || sl;
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi, 31, &ahi) || sh;
  else
    // post-accumulate fractional shift mode
    addlo = -b.h0 * c.h0;
    addhi = -b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1
: ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1
: ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= sl | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMSUB2HF r1,r2,r3                ; Signed fractional multiply-
                                ;subtract two 16-bit vectors r2
                                ;and r3. return result in r1

```

Syntax and Encoding

Instruction Code

VMSUB2HF	a,b,c	00110 bbb 00 000100 0 BBB CCCCC AAAAAA
VMSUB2HF	a,b,u6	00110 bbb 01 000100 0 BBB uuuuuu AAAAAA
VMSUB2HF	b,b,s12	00110 bbb 10 000100 0 BBB ssssss SSSSSS
VMSUB2HF<.cc>	b,b,c	00110 bbb 11 000100 0 BBB CCCCC 0QQQQQ
VMSUB2HF<.cc>	b,b,u6	00110 bbb 11 000100 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMSUB2HFR

Function

Two-way signed multiplication and subtraction of two 16-bit fractional vectors. The result is rounded and saturated.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16(RND16((acchi -= (b.h1*c.h1))<<1));
a.h0 = SAT16(RND16((acclo -= (b.h0*c.h0))<<1));
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VMSUB2HFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and subtract the product from the high accumulator and store the result in the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c`, subtract the product from the low accumulator, and store the result in the low accumulator. Return the results as a two-way rounded 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* VMSUB2HFR*/
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = -b.h0 * c.h0 << 1;
        addhi = -b.h1 * c.h1 << 1;
        acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        sl = sat(-(b.h0 * c.h0 << 1), 31, &addlo);
        sh = sat(-(b.h1 * c.h1 << 1), 31, &addhi);
        acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo) || sl;
        acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi) || sh;
    else
        // post-accumulate fractional shift mode
        addlo = -b.h0 * c.h0;
        addhi = -b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMSUB2HFR r1,r2,r3          ; Signed fractional multiply-
                             ;subtract two 16-bit vectors r2 and
                             ;r3. return rounded result in r1

```

Syntax and Encoding

Instruction Code

VMSUB2HFR	a,b,c	00110bbb00000110BBBCCCCCAAAAAA
VMSUB2HFR	a,b,u6	00110bbb010000110BBBuuuuuuAAAAAA
VMSUB2HFR	b,b,s12	00110bbb10000110BBBssssssSSSSSS
VMSUB2HFR<.cc>	b,b,c	00110bbb110000110BBBCCCCC0QQQQQ
VMSUB2HFR<.cc>	b,b,u6	00110bbb110000110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMSUB2HNFR

Function

Two-way signed multiplication and subtraction of two 16-bit fractional vectors. The result is saturated, and returned without a fractional shift.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16(RND16(acchi - (b.h1*c.h1)));
a.h0 = SAT16(RND16(acclo - (b.h0*c.h0)));
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VMSUB2HNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and subtract the product from the high accumulator and store the result in the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c`, subtract the product from the low accumulator, and store the result in the low accumulator. Return the results as a two-way rounded 16-bit fractional vector. The result is not shifted by one bit. This instruction does not modify any flags.

Pseudo Code

```

// note: accumulators modeled as 128b integers
addlo = -b.h0 * c.h0;
addhi = -b.h1 * c.h1;
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= sl | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMSUB2HNFR r1,r2,r3          ; Signed fractional multiply-
                             ;subtract two 16-bit vectors r2 and
                             ;r3. return rounded result in r1

```

Syntax and Encoding

Instruction Code

VMSUB2HNFR	a,b,c	00110 bbb 00 0100011 BBB CCCCC AAAAAA
VMSUB2HNFR	a,b,u6	00110 bbb 01 0100011 BBB uuuuuu AAAAAA

VMSUB2HNFR	b, b, s12	00110 bbb 10 010001 1 BBB ssssss SSSSSS
VMSUB2HNFR<.cc>	b, b, c	00110 bbb 11 010001 1 BBB CCCCC 0QQQQQ
VMSUB2HNFR<.cc>	b, b, u6	00110 bbb 11 010001 1 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VNEG2H

Function

Negate the two 16-bit vectors of an operand and store the negated value in the destination operand.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = -c.h1;
b.h0 = -c.h0;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VNEG2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Negate the two 16-bit vectors of an operand and store the negated value in the destination operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
b1 = 0 - c1;
bh = 0 - ch;
b = (b1 & 0x0000_ffff) | (bh << 16);
```

/* VNEG2H/*

Assembly Code Example

```
VNEG2H r1,r2          ; Negate vector r2 and return result in r1
```

Syntax and Encoding

Instruction Code		
VNEG2H	b, c	00101 bbb 00 101111 0 BBB CCCCC 101010
VNEG2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101010
VNEG2H	0, c	00101 110 00 101111 0 111 CCCCC 101010
VNEG2H	0, limm	00101 110 00 101111 0 111 111110 101010
VNEG2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101010
VNEG2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VNEGS2H

Function

Negate the two 16-bit vectors of an operand and store the saturated negated value in the destination operand.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = SAT16(-c.h1);
b.h0 = SAT16(-c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VNEGS2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Negate the two 16-bit vectors of an operand and store the saturated negated value in the destination operand. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

c1 = c & 0x0000_ffff;                                     /* VNEGS2H/*
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)0 - c1, 15, &b1);
sh = sat((sint128)0 - ch, 15, &bh);
DSP_CTRL.SAT |= s1 | sh;
b = (b1 & 0x0000_ffff) | (bh << 16);

```

Assembly Code Example

```
VNEGS2H r1,r2      ; Negate vector r2 and return saturated result in r1
```

Syntax and Encoding

Instruction Code

VNEGS2H	b, c	00101 bbb 00 101111 0 BBB CCCCC 101011
VNEGS2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101011
VNEGS2H	0, c	00101 110 00 101111 0 111 CCCCC 101011
VNEGS2H	0, limm	00101 110 00 101111 0 111 111110 101011
VNEGS2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101011
VNEGS2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101011

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VNORM2H

Function

Two-way 16-bit vector normalization.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = norm(c.h1);
b.h0 = norm(c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VNORM2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Normalize a two-way 16-bit vector to return a vector of normalized constants. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

short normh(short c) {
    int i;
    for (i = 0; (i < 16) && ((c >> i) != 0) && ((c >> i) != -1); i++);
    return 15 - i;
}
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
b1 = normh(c1);
bh = normh(ch);
b = (b1 & 0x0000_ffff) | (bh << 16);

```

/*
VNORM2H/*

Assembly Code Example

```

VNORM2H r1,r2      ; Normalize each 16-bit vector in r2 and return
                   ; result in r1

```

Syntax and Encoding

Instruction Code

VNORM2H	b, c	00101 bbb 00 101110 0 BBB CCCCC 101100
VNORM2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101100
VNORM2H	0, c	00101 110 00 101111 0 111 CCCCC 101100
VNORM2H	0, limm	00101 110 00 101111 0 111 111110 101100
VNORM2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101100
VNORM2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HL

Function

Compose the destination operand from the lower 16-bits of the source operands.

Extension Group

DSP vector unpacking

Operation

```
a.h0 = c.h0;
a.h1 = b.h0;
```

Instruction Format

op a,b,c

Syntax Example

VPACK2HL a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose the destination operand from the lower 16-bits of the source operands. This instruction does not modify any flags.

Pseudo Code

```
a = ((b & 0x0000ffff) << 16) | (c & 0x0000ffff) /* VPACK2HL*/
```

Assembly Code Example

```
VPACK2HL r0,r1 ; Pack lower 16 bits from r1 and r2 into r0
```

Syntax and Encoding

		Instruction Code
VPACK2HL	a, b, c	00101bbb001010010BBBCCCCCAAAAAA
VPACK2HL	a, b, u6	00101bbb011010010BBBuuuuuuAAAAAA
VPACK2HL	b, b, s12	00101bbb101010010BBBssssssSSSSSS
VPACK2HL<.cc>	b, b, c	00101bbb111010010BBBCCCCC0QQQQQ
VPACK2HL<.cc>	b, b, u6	00101bbb111010010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HM

Function

Compose the destination operand from the higher 16-bits of the source operands.

Extension Group

DSP vector unpacking

Operation

```
a.h0 = c.h1;
a.h1 = b.h1;
```

Instruction Format

op a,b,c

Syntax Example

VPACK2HM a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose the destination operand from the higher 16-bits of the source operands. This instruction does not modify any flags.

Pseudo Code

```
a = ((b & 0x0000ffff) << 16) | (c & 0x0000ffff) /* VPACK2HM*/
```

Assembly Code Example

```
VPACK2HM r0,r1 ; Pack higher 16 bits from r1 and r2 into r0
```

Syntax and Encoding

		Instruction Code
VPACK2HM	a, b, c	00101bbb001010011BBBCCCCCAAAAAA
VPACK2HM	a, b, u6	00101bbb011010011BBBuuuuuuAAAAAA
VPACK2HM	b, b, s12	00101bbb101010011BBBssssssSSSSSS
VPACK2HM<.cc>	b, b, c	00101bbb111010011BBBCCCCC0QQQQQ
VPACK2HM<.cc>	b, b, u6	00101bbb111010011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HBL

Function

Pack the LSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the lower 16 bits of the destination.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = 0;
b.b2 = 0;
b.b1 = c.b2;
b.b0 = c.b0;
```

Instruction Format

op b, c

Syntax Example

VPACK2HBL b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Extract the two least significant bytes from a dual packed 16-bit elements of an operand, and return the extracted bytes in the lower two bytes of the destination operand. This instruction does not modify any flags.

Pseudo Code

```
b = ((c & 0x00ff0000) >> 8) | (c & 0x000000ff) /* VPACK2HBL*/
```

Assembly Code Example

```
VPACK2HBL r0,r1      ; Pack two lower significant bytes from r1
                      ; into lower part of a word in r0
```

Syntax and Encoding

Instruction Code

VPACK2HBL	b, c	00101bbb001011110BBBCCCCC011100
VPACK2HBL	b, u6	00101bbb011011110BBBuuuuuu011100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HBLF

Function

Pack the MSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the lower 16 bits of the destination.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = 0;
b.b2 = 0;
b.b1 = c.b3;
b.b0 = c.b1;
```

Instruction Format

op b, c

Syntax Example

VPACK2HBLF b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Extract the two most significant bytes from a the two 16-bit elements of the source operand (c), and return the extracted bytes in the lower two bytes of the destination operand (b). This instruction does not modify any flags.

Pseudo Code

```
b = ((c & 0xff000000) >> 16) | ((c & 0x0000ff00) >> 8) /* VPACK2HBLF*/
```

Assembly Code Example

```
VPACK2HBLF r0,r1      ; Pack two most significant bytes from r1
                       ; into lower part of a word in r0
```

Syntax and Encoding

Instruction Code

VPACK2HBLF	b, c	00101bbb001011110BBBCCCCC011110
VPACK2HBLF	b, u6	00101bbb011011110BBBuuuuuu011110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HBM

Function

Pack the LSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the higher 16 bits of the destination.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = c.b2;
b.b2 = c.b0;
b.b1 = 0;
b.b0 = 0;
```

Instruction Format

op b, c

Syntax Example

VPACK2HBM b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Extract the two most significant bytes from the 16-bit elements of the source operand (c), and return the extracted bytes in the higher two bytes of the destination operand (b). This instruction does not modify any flags.

Pseudo Code

```
b = ((c & 0x00ff0000) << 8) | ((c & 0x000000ff) << 16) /* VPACK2HBM*/
```

Assembly Code Example

```
VPACK2HBM r0,r1      ; Pack two lower significant bytes from r1
                      ; into upper part of a word in r0
```

Syntax and Encoding

Instruction Code

VPACK2HBM	b, c	00101bbb001011110BBBCCCCC011101
VPACK2HBM	b, u6	00101bbb011011110BBBuuuuuu011101

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HBMF

Function

Pack the MSBs of a 2x16 bit vectors into a 2x8 bit vector. Result is stored in the higher 16 bits of the destination.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = c.b3;
b.b2 = c.b1;
b.b1 = 0;
b.b0 = 0;
```

Instruction Format

op b, c

Syntax Example

VPACK2HBMF b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Extract the two most significant bytes from a dual packed 16b half-word (c). Return the two bytes in the upper two bytes of operand b. This instruction does not modify any flags.

Pseudo Code

```
b = (c & 0xff000000) | ((c & 0x0000ff00) << 8) /* VPACK2HBMF*/
```

Assembly Code Example

```
VPACK2HBMF r0,r1    ; Pack two most significant bytes from r1
                    ; into upper part of a word in r0
```

Syntax and Encoding

Instruction Code

VPACK2HBMF	b, c	00101 bbb 00 101111 0 BBB CCCCC 011111
VPACK2HBMF	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 011111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPERM

Function

Perform byte permutation with zero or sign extension.

Extension Group

DSP vector unpacking

Operation

$$a = \text{vperm}(b, c)$$

Instruction Format

op a, b, c

Syntax Example

VPERM a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Swap bytes in a word and optionally zero or sign extend a byte.

Pseudo Code

```

// get three bit byte index value                                /* VPERM*/
index[0] = (c>>0) & 0x7;
index[1] = (c>>3) & 0x7;
index[2] = (c>>6) & 0x7;
index[3] = (c>>9) & 0x7;
// get one bit extend enable per byte
ext[0] = (c >> 12) & 1
ext[1] = (c >> 13) & 1
ext[2] = (c >> 14) & 1
ext[3] = (c >> 15) & 1
// do for each byte
sgn = 0;
a = 0;
for (i = 0; < 4; i++) {
    if (ext[i]) {
        // extend
        if ((index[i] & 1) == 0) {
            // zero extend
            byte = 0;
        } else {
            // sign extend
            byte = sgn ? 0xFF : 0x00;
        }
    } else if (index[i] < 4) {
        // copy byte from location index[i]
        byte = (b >> (8*index[i])) & 0xff;
    } else {
        // byte is out of range
        byte = 0
    }
    a = a | (byte << (8*i)); // add byte to output
    sgn = (byte & 0x80) != 0; // store sign for next
iteration
}

```

Assembly Code Example

```

VPERM r0,r1,r2           ; Permute the bytes from r2 as defined in r1
                          ; and write-back the result in r0

```

Syntax and Encoding

Instruction Code

VPERM	a,b,c	00101 bbb 00 101110 0 BBB CCCCC AAAAAA
VPERM	a,b,u6	00101 bbb 01 101110 0 BBB uuuuuu AAAAAA

VPERM	b, b, s12	00101 bbb 10 101110 0 BBB ssssss SSSSSS
VPERM<.cc>	b, b, c	00101 bbb 11 101110 0 BBB CCCCC0 QQQQQ
VPERM<.cc>	b, b, u6	00101 bbb 11 101110 0 BBB uuuuu1 QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VREP2HL

Function

Compose a dual 16-bit vector by replicating the least-significant 16 bits of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.h1 = c.h0;
b.h0 = c.h0;
```



Note

`h0` and `h1` represent the 16-bit elements of an operand.

Instruction Format

op b, c

Syntax Example

VREP2HL b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by replicating the least-significant 16 bits of the `c` operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b = (c << 16) | (c & 0x0000_ffff); /* VREP2HL*/
```

Assembly Code Example

```
VREP2HL r1,r2      ; replicate lower 16-bits of r2 into r1.
```

Syntax and Encoding

Instruction Code		
VREP2HL	b, c	00101bbb001011110BBBCCCCC100010
VREP2HL	b, limm	00101bbb001011110BBB111110100010
VREP2HL	0, c	00101110001011110111CCCCC100010
VREP2HL	0, limm	00101110001011110111111110100010
VREP2HL	b, u6	00101bbb011011110BBBuuuuuu100010
VREP2HL	0, u6	00101110011011110111uuuuuu100010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VREP2HM

Function

Compose a dual 16-bit vector by replicating the most-significant 16 bits of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.h1 = c.h1;
b.h0 = c.h1;
```



Note

`h0` and `h1` represent the 16-bit elements of an operand.

Instruction Format

op b, c

Syntax Example

VREP2HM b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by replicating the most-significant 16 bits of the `c` operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b = (c & 0xffff_0000) | ((c >>16) & 0x0000_ffff);    /* VREP2HM*/
```

Assembly Code Example

```
VREP2HM r1,r2      ; replicate higher 16-bits of r2 into r1.
```

Syntax and Encoding

Instruction Code		
VREP2HM	b, c	00101bbb001011110BBBCCCCC100011
VREP2HM	b, limm	00101bbb001011110BBB111110100011
VREP2HM	0, c	00101110001011110111CCCCC100011
VREP2HM	0, limm	00101110001011110111111110100011
VREP2HM	b, u6	00101bbb011011110BBBuuuuuu100011
VREP2HM	0, u6	00101110011011110111uuuuuu100011

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSEXT2BHL

Function

Compose a dual 16-bit vector by sign-extending the lower two bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.h1 = sext(c.b1);
b.h0 = sext(c.b0);
```



Note

b0 and b1 represent the 8-bit elements of an operand. h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b, c

Syntax Example

VSEXT2BHL a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by sign-extending the lower two bytes of the c operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = ((c << 24) >> 24) & 0x0000_ffff) | (((c & 0x0000_ff00) << 16) >> 8); /* VSEXT2BHL*/
```

Assembly Code Example

```
VSEXT2BHL r1,r2      ; sign extend two lower bytes from r2 into 16-bit
                      ; words in r1
```

Syntax and Encoding

Instruction Code		
VSEXT2BHL	b, c	00101bbb001011110BBBCCCCC100110
VSEXT2BHL	b, limm	00101bbb001011110BBB111110100110
VSEXT2BHL	0, c	00101110001011110111CCCCC100110
VSEXT2BHL	0, limm	00101110001011110111111110100110
VSEXT2BHL	b, u6	00101bbb011011110BBBuuuuuu100110
VSEXT2BHL	0, u6	00101110011011110111uuuuuu100110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSEXT2BHM

Function

Compose a dual 16-bit vector by sign-extending the higher two bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.h1 = sext(c.b3);
b.h0 = sext(c.b2);
```



Note

b2 and b3 represent the 8-bit elements of an operand. h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VSEXT2BHM b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by sign-extending the higher two bytes of the c operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = ((c << 8) >> 24) & 0x0000_ffff | ((c & 0xff00_0000) >> 8); /* VSEXT2BHM*/
```


Assembly Code Example

```
VSEXT2BHM r1,r2      ; sign extend two higher bytes from r2 into 16-bit
                      ; words in r1
```

Syntax and Encoding

Instruction Code		
VSEXT2BHM	b, c	00101 bbb 00 101111 0 BBB CCCCC 100111
VSEXT2BHM	b, limm	00101 bbb 00 101111 0 BBB 111110 100111
VSEXT2BHM	0, c	00101 110 00 101111 0 111 CCCCC 100111
VSEXT2BHM	0, limm	00101 110 00 101111 0 111 111110 100111
VSEXT2BHM	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100111
VSEXT2BHM	0, u6	00101 110 01 101111 0 111 uuuuuu 100111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUB2H

Function

Dual 16-bit vector subtraction.

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Each pair of elements from b and c is subtracted to form two 16-bit differences. These differences are assigned to the destination register, if defined.

This instruction does not modify any flags. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if(cc) {                                     /* VSUB2H*/
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 - c.h1;
}
```

Assembly Code Example

```
VSUB2H r1,r2,r3                          ; dual SIMD 16-bit subtraction
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUB2H	a, b, c	00101bbb000101010BBBCCCCCAAAAAA
VSUB2H	a, b, u6	00101bbb010101010BBBuuuuuuAAAAAA
VSUB2H	b, b, s12	00101bbb100101010BBBssssssSSSSSS
VSUB2H<.cc>	b, b, c	00101bbb110101010BBBCCCCC0QQQQQ
VSUB2H<.cc>	b, b, u6	00101bbb110101010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBADD2H

Function

Dual 16-bit vector subtraction and addition.

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 + c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUBADD2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit *b* and *c* operands specify vectors containing two 16-bit elements. The lower 16-bit elements from *b* and *c* are subtracted to form a 16-bit difference. The higher 16-bit elements are added to form a 16-bit sum. The difference and sum are assigned to the destination register.

This instruction does not modify any flags. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if(cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 + c.h1;
}
```

/* VSUBADD2H*/

Assembly Code Example

```
VSUBADD2H r1,r2,r3    ; dual SIMD 16-bit subtraction and addition of r2
                      ; and r3
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUBADD2H	a, b, c	00101bbb000101110BBBCCCCCAAAAA
VSUBADD2H	a, b, u6	00101bbb010101110BBBuuuuuuAAAAA
VSUBADD2H	b, b, s12	00101bbb100101110BBBssssssSSSSSS
VSUBADD2H<.cc>	b, b, c	00101bbb110101110BBBCCCCC0QQQQQ
VSUBADD2H<.cc>	b, b, u6	00101bbb110101110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUB2H

Function

Dual 16-bit vector subtraction.

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Each pair of elements from `b` and `c` is subtracted to form two 16-bit differences. These differences are assigned to the destination register, if defined.

This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                /* VSUB2H*/
bh = (b >> 16) & 0x0000_ffff;
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
a1 = b1 - c1;
ah = bh - ch;
a = (a1 & 0x0000_ffff) | (ah << 16);

```

Assembly Code Example

```
VSUB2H r1,r2,r3 ; dual SIMD 16-bit subtraction
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

VSUB2H	a, b, c	00101bbb000101010BBBCCCCCAAAAAA
VSUB2H	a, b, u6	00101bbb010101010BBBuuuuuuAAAAAA
VSUB2H	b, b, s12	00101bbb100101010BBBssssssSSSSSS
VSUB2H<.cc>	b, b, c	00101bbb110101010BBBCCCCC0QQQQQ
VSUB2H<.cc>	b, b, u6	00101bbb110101010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBS2H

Function

Dual 16-bit vector subtraction. The result is saturated.

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = SAT16(b.h0 - c.h0);
    a.h1 = SAT16(b.h1 - c.h1);
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUBS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Each pair of elements from b and c is subtracted to form two 16-bit differences. These saturated differences are assigned to destination register.

The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                /* VSUBS2H*/
bh = (b >> 16) & 0x0000_ffff;
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)b1 - c1, 15, a1);
sh = sat((sint128)bh - ch, 15, ah);
a = (a1 & 0x0000_ffff) | (ah << 16);
DSP_CTRL.SAT |= s

```

Assembly Code Example

```

VSUBS2H r1,r2,r3                                ; dual SIMD 16-bit subtraction
                                                ;and store the saturated result

```

Syntax and Encoding

Instruction Code		
VSUBS2H	a,b,c	00101bbb000101011BBBCCCCCAAAAAA
VSUBS2H	a,b,u6	00101bbb010101011BBBuuuuuuAAAAAA
VSUBS2H	b,b,s12	00101bbb100101011BBBssssssSSSSSS
VSUBS2H<.cc>	b,b,c	00101bbb110101011BBBCCCCC0QQQQQ
VSUBS2H<.cc>	b,b,u6	00101bbb110101011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUB4B

Function

Four-way eight-bit vector subtraction.

Extension Group

DSP vector arithmetic

Operation

$$\begin{aligned} a.b3 &= b.b3 - c.b3; \\ a.b2 &= b.b2 - c.b2; \\ a.b1 &= b.b1 - c.b1; \\ a.b0 &= b.b0 - c.b0; \end{aligned}$$


Note

$b0$, $b1$, $b2$, and $b3$ represent the 8-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VSUB4B a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Pair-wise byte subtraction of the b and c operands, and the results are stored in the a operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

b1l = b & 0x0000_00ff;
b1h = b & 0x0000_ff00;
bhl = b & 0x00ff_0000;
bhh = b & 0xff00_0000;
c1l = c & 0x0000_00ff;
c1h = c & 0x0000_ff00;
chl = c & 0x00ff_0000;
chh = c & 0xff00_0000;
all = b1l - c1l;
alh = b1h - c1h;
ahl = bhl - chl;
ahh = bhh - chh;
a = (all & 0x0000_00ff) | (alh & 0x0000_ff00) | (ahl & 0x00ff_0000) |
    ahh;

```

/* VSUB4B/*

Assembly Code Example

```

VSUB4B r0,r1,r2      ; Subtract two vectors r1 and r2, and store the
                      result in ;r0;

```

Syntax and Encoding

Instruction Code

VSUB4B	a,b,c	00101 bbb 00 1001010 BBB CCCCC AAAAAA
VSUB4B	a,b,u6	00101 bbb 01 1001010 BBB uuuuuu AAAAAA
VSUB4B	b,b,s12	00101 bbb 10 1001010 BBB ssssss SSSSSS
VSUB4B<.cc>	b,b,c	00101 bbb 11 1001010 BBB CCCCC 0QQQQQ
VSUB4B<.cc>	b,b,u6	00101 bbb 11 1001010 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBADDS2H

Function

Dual 16-bit vector subtraction and addition. The results are saturated.

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = SAT16 (b.h0 - c.h0);
    a.h1 = SAT16 (b.h1 + c.h1);
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUBADDS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit `b` and `c` operands specify vectors containing two 16-bit elements. The lower 16-bit elements from `b` and `c` are subtracted to form a 16-bit difference. The higher 16-bit elements are added to form a 16-bit sum. The difference and sum are assigned to the destination register.

The saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix if the results of this instruction saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                /* VSUBADDS2H*/
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)b1 - cl, 15, al);
sh = sat((sint128)bh + ch, 15, ah);
a = (al & 0x0000_ffff) | (ah << 16);
DSP_CTRL.SAT |= s

```

Assembly Code Example

```

VSUBADDS2H r1,r2,r3    ; dual SIMD 16-bit subtraction and addition and
                        ;store the saturated result

```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

VSUBADDS2H	a, b, c	00101bbb000101111BBBCCCCCAAAAAA
VSUBADDS2H	a, b, u6	00101bbb010101111BBBuuuuuuAAAAAA
VSUBADDS2H	b, b, s12	00101bbb100101111BBBssssssSSSSSS
VSUBADDS2H<.cc>	b, b, c	00101bbb110101111BBBCCCCC0QQQQQ
VSUBADDS2H<.cc>	b, b, u6	00101bbb110101111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

Part 5

Appendixes

In this part:

- [ISA Version Differences](#)
- [Implementation-dependent Behavior](#)
- [Auxiliary Functions for DSP and XY Extensions](#)
- [Examples](#)
- [ARConnect Commands Details](#)

A

ISA Version Differences

This appendix summarizes the differences between ARCV2, ARCompact for ARC 600, and ARCompact for ARC 700.

The following are the most significant differences:

- Major opcode 0x08, previously listed in the ARCompact PRM as “User 32-bit extension instructions,” is reassigned to encode 16-bit ARC extensions.
- Major opcodes 0x09, 0x0A, and 0x0B, previously listed in the ARCompact PRM as “ARC market-specific extension instructions (32-bit),” are assigned in ARCV2 as 16-bit extension formats.
- The format of major opcode 0x0E has been revised to permit twice as many operators to be encoded, at the cost of reducing the width of the ‘h’ register operand from 6 to 5 bits. Instructions from ARCompact that used this format are not binary compatible with ARCV2, and *vice versa*.
- The major opcode 0x0F, [0x00-0x1F], General Register Format, now has a re-assignment of sub-opcode 0x0C, from MUL_S to MPY_S. When MPY_OPTION is set to include 32-bit multiply instructions, this encoding is used as a synonym for the 32-bit signed MPY instruction. When MPY_OPTION defines that either no multiply instructions are present or only 16x16 multiply operations are present, any use of sub-opcode 0x0C raises an Instruction Error exception.
- The J.F [ILINKn] instruction is not supported in ARCV2. The RTIE instruction handles all interrupt and exception returns. Use of the .F notation in a J instruction is silently ignored because this bit is reserved in these formats.
- The ARCV2 architecture has only one trap instruction. The TRAP0 instruction of the V1-700 architecture is replaced by the SWI instruction found in the V1-600 architecture.
- The ARCV2 architecture alters the semantics of SWI compared with the V1-700 TRAP0/SWI (which one could argue is not defined correctly). The ARCV2 architecture does not commit the SWI instruction, and therefore sets the ERET auxiliary register to the address of the SWI instruction itself.
- The ARCV2 architecture defines an SWI_S instruction to allow software breakpoints to be inserted in place of the 16-bit instructions.
- The ARC EM processors with ARCOVER ≥ 0x44 always include the EFA register. The ARC EM family of processors with ARCOVER < 0x44 include the EFA register only when any of the following protections are configured: MMU, MPU, code protection, stack checking, ECC, or parity protection. The EFA register is always included in the ARC HS family of processors and processors based on the V1-700 architecture whereas the V1-600 architecture does not define this register.

- The ARCV2 architecture does not include the BTA_L1 or BTA_L2 aux registers. The V1-700 architecture includes these registers. The V1-600 architecture does not define these registers.

Table A-1 and Table A-2 describe the known differences in the baseline and optional architecture features between ARCV2, ARCompact for ARC 600, and ARCompact for ARC 700.

Table A-1 Baseline Differences: ARC 600, ARC 700, and ARCV2

Baseline ISA Features	ARC 600	ARC 700	ARCV2
RTIE	No	Yes	Yes
J.F [ilink] returns from interrupt	Yes	Yes	No (use RTIE)
EX	No	Yes	Yes
TRAP0	No	Yes	No
TRAP_S	No	Yes	Yes
SWI	Yes	No	Yes
AEX	No	No	Yes
SETI	No	No	Yes
CLRI	No	No	Yes
SWI_S	No	No	Yes
BRK	No	Yes	Yes
DIVAW	Yes	Yes	No
Saturating ALU and shift operations, Condition codes 0x10, 0x11	No	Yes	No
UNIMP_S	No	Yes	Yes
User / Kernel modes	No	Yes	Yes
Full exception handling Privileged instructions Protected state	No	Yes	Yes
BTA	No	Yes	Yes
BTA_L1	No	Optional	No
BTA_L2	No	Optional	No
EFA auxiliary register	No	Yes	Baseline
STATUS register (obsolete)	Yes Not in 601	Yes	No

Table A-1 Baseline Differences: ARC 600, ARC 700, and ARCV2 (Continued)

Baseline ISA Features	ARC 600	ARC 700	ARCV2
Semaphore register	Yes Not in 601	No	No
16 Register Option	Yes	No	Yes
Exception on access to unimplemented core register?	No	Yes	Yes
Exception on LD, EX, POP, or other multi-cycle operation with LP_COUNT as destination?	No	Yes Privilege Violation	Yes Instruction Error exception
Protected access to LP_COUNT?	No	Yes	Yes
Exception on attempting to write to PCL?	No (undefined behavior)	Yes	Yes
PCL usable as source registers in BRcc/BBITn?	No	Yes	Yes
Reading a non-existent auxiliary register	Returns the IDENTITY register value	Instruction Error exception	Instruction Error exception
Writing a non-existent auxiliary register	Write is ignored	Instruction Error exception	Instruction Error exception
Reading a write-only aux register	Returns IDENTITY register value	Instruction Error exception	Instruction Error exception
Writing a read-only aux register	Write is ignored	Instruction Error exception	Instruction Error exception
Reading a non-existent BCR register	Returns 0	Return 0 in kernel mode, Privilege Violation Exception in user mode	Return 0 in kernel mode, Privilege Violation Exception in user mode
Load with L IMM destination (prefetch)	No	Yes	Yes
Load/Store data size mode ZZ = 11	Undefined behavior	Instruction Error exception	Instruction Error exception
Load-extend mode X = 1, when size ZZ = 00 (32-bit word)	Ignored	Instruction Error exception	Instruction Error exception
Illegal combinations of fields	Ignored	Instruction Error exception	Instruction Error exception
Use of undefined condition codes	Always returns FALSE	Instruction Error exception	Instruction Error exception

Table A-1 Baseline Differences: ARC 600, ARC 700, and ARCV2 (Continued)

Baseline ISA Features	ARC 600	ARC 700	ARCV2
BRcc sub-opcodes 0x6 to 0xD	Not defined by the PRM	Instruction Error exception	Used to encode static branch predictions
Branch in delay slot, LPcc in delay slot, RTIE in delay slot, LIMM in delay slot instruction	Undefined behavior!	Illegal Instruction Sequence exception	Illegal Instruction Sequence exception
FLAG is a serializing instruction?	No	Yes	Yes
KFLAG instruction supported?	No	No	Yes
SLEEP has placement restrictions?	Yes	No	No
LR or SR with operand mode 0x3	Undefined behavior	Instruction Error Exception	Instruction Error Exception
w6 operand format for storing an immediate 6-bit value	No	No	Yes
Misaligned data memory exceptions	Detected, but not prevented	Protection Violation exception	Misaligned Access exception (new)
Misaligned data memory accesses	Not supported	Not supported	Supported

Table A-2 ISA Option Differences: ARC 600, ARC 700, and ARCV2

Optional ISA Features	ARC 600	ARC 700	ARCV2
EX	No	Baseline	Baseline
EFA auxiliary register	No	Baseline	Baseline
MUL64 MULU64 MUL64_S MLO, MMID, MHI, and MULHI extension core registers	Optional	No	No
MPY MPYU MPYH MPYHU	No	Optional	Optional
MPY_S	No	No	Optional
MPYW MPYUW	Optional	No	Optional

Table A-2 ISA Option Differences: ARC 600, ARC 700, and ARCV2 (Continued)

Optional ISA Features	ARC 600	ARC 700	ARCV2
DIV DIVU REM REMU	No	No	Optional
Divide-by-zero exception and STATUS32.DZ	No	No	Optional
NORM NORMW	Optional	Optional	Optional
SWAP	Optional	Optional	Optional
SWAPE	No	Baseline	Optional
LDI, LDI_S	No	No	Optional
LSL16, LSR16	No	No	Optional
Barrel shifter	Optional	Optional	Optional
Shift-assist instructions ASR16, ASR8, LSR8, LSL8, ROL8, ROR8	No	No	Optional
SETcc instructions: SETEQ, SETNE, SETLT, SETGE, SETLO, SETHS, SETLE, SETGT	No	No	Optional
ENTER_S, LEAVE_S	No	No	Optional
EI_S	No	No	Optional
JLI_S	No	No	Optional
BI, BIH	No	No	Optional
FFS, FLS	No	No	Optional

Table A-3 ISA Features: ARC EM and ARC HS

ISA Feature	ARC EM	ARC HS3x	ARC HS4x
Multiply and accumulate extension instructions: QMPYH, QMPYHU, DMPYWH, DMPYWHU, QMACH, QMACHU, DMACWH, DMACWHU,	No	Optional Select these instructions using the mpy_option = {7,8,9} option.	Optional Select these instructions using the mpy_option = {7,8,9} option.
Vector addition and subtraction instructions: VADD4H, VSUB4H, VADDSUB4H, VSUBADD4H, VADD2, VSUB2, VADDSUB, VSUBADD	No	Optional Select these instructions using the mpy_option = {7,8,9} option.	Optional Select these instructions using the mpy_option = {7,8,9} option.
DSP Extension Instructions	Optional Select these instructions using the has_dsp==1 or dsp_complex==1	No	Optional Select these instructions using the has_dsp==true
DSP Extension Instructions for Complex Numbers	Optional Select these instructions using the dsp_complex==1	No	Optional Select these instructions using the has_dsp==true
DSP wide extension instructions	No	No	Optional Select these instructions using the dsp_wide==true
LPcc: branch or jump in last position within loop body	Affects loop count	Disallowed: triggers illegal instruction sequence exception (see LPcc instruction)	Disallowed: triggers illegal instruction sequence exception (see LPcc instruction)
Double-word 64-bit operations	No	Optional, Enable/disable 64-bit operations using ll64_option = {false,true}. LDD and STD are load and store operations for 64-bit data. When atomic operation are also enabled, LLOCKD and SCONDD instructions are enabled.	Optional, Enable/disable 64-bit operations using ll64_option = {false,true}. LDD and STD are load and store operations for 64-bit data. When atomic operation are also enabled, LLOCKD and SCONDD instructions are enabled.

Table A-3 ISA Features: ARC EM and ARC HS

ISA Feature	ARC EM	ARC HS3x	ARC HS4x
r58 and r59	Extension core registers When <code>has_dsp==true</code> , r58 is used as an Accumulator Low register in little-endian systems and Accumulator High in big-endian systems, and r59 is used as an Accumulator High register in little-endian systems and Accumulator Low register in big-endian systems. When a floating-point unit is included and the FSMADD and FSMSUB instructions are enabled, the two extension core registers (r58 and r59) are included and comprise the third operand.	When <code>mpy_option > 6</code> , r58 is used as an Accumulator Low register in little-endian systems and Accumulator High in big-endian systems, and r59 is used as an Accumulator High register in little-endian systems and Accumulator Low register in big-endian systems. When <code>mpy_option > 6</code> or <code>fpu_fma_option</code> is defined, two extension core registers (r58 and r59) are included and comprise the 64-bit Accumulator (ACCH, ACCL).	When <code>mpy_option > 6</code> , r58 is used as an Accumulator Low register in little-endian systems and r59 is used as an Accumulator High register. When <code>mpy_option > 6</code> , <code>has_dsp==1</code> , or <code>fpu_fma_option</code> is defined, two extension core registers (r58 and r59) are included and comprise the 64-bit Accumulator (ACCH, ACCL).
IFQUEUE_BUILD auxiliary register	Included in this ISA and configurable.	Not included in this ISA.	Not included in this ISA.
<code>code_density_option</code>	Optional	Yes	Yes
<code>bit_scan_option</code>	Optional	Yes	Yes
<code>swap_option</code>	Optional	Yes	Yes
<code>byte_order</code>	little and big endian	little and big endian	little endian only

Table A-3 ISA Features: ARC EM and ARC HS

ISA Feature	ARC EM	ARC HS3x	ARC HS4x
ATOMIC_OPTION	No	Optional; When <code>atomic_option==true</code> , the following instructions are supported: LLOCK, SCOND, WLFC Further when <code>ll64_option==1</code> , the LLOCKD and SCONDD instructions are supported.	Optional; When <code>atomic_option==true</code> , the following instructions are supported: LLOCK, SCOND, WLFC Further when <code>ll64_option==true</code> , the LLOCKD and SCONDD instructions are supported.
Floating-point instructions	Yes	Yes	Yes
IC_FEATURE_LEVEL	Optional; allowed values are 0, 1, 2	Fixed at 2.	Fixed at 2.
DC_FEATURE_LEVEL	Optional; allowed values are 0, 1, 2	Fixed at 2.	Fixed at 2.
Degree of associativity for data caches (DC_WAYS)	Optional; allowed values are 0, 1, 2	Fixed at 2.	Fixed at 2.
Support for Memory management Unit	No	Yes	Yes
Support for L2 cache	No	Yes	Yes
Baseline instructions: PREFETCHW, DSYNC, DMB	No	Yes	Yes
Support for Shared Coherency Unit	No	Yes	Yes
Support for the Vector Unit	No	No	No
Multi-core cluster component	No	Yes	Yes
Bitstream instructions	Yes	No	No
XY instructions: MODIF, MODAPP	Yes	No	No
Address Generation Unit	Yes	No	Yes
FastMath extensions	No	Yes	Yes
Support for shared memory configuration between CSM and L2 cache	No	Yes	Yes
Conversion instructions: FS2H and FH2S	No	No	Yes

B

Implementation-dependent Behavior

B.1 EM Exception Handling

B.1.1 ECR User-mode Setting in the ARC EM Family of Cores

The U field, located in bit 30 of the ECR auxiliary register, indicates that although the processor was in user mode when an exception occurred, kernel privileges were in force at the time. This can happen when an interrupt prolog or epilog pushes or pops registers to or from the kernel stack when a user-mode process is interrupted. Kernel privileges are applied to all interrupt prolog memory operations that access the kernel stack. Upon completion of register push operations during an interrupt prolog, the processor transitions into kernel mode, but the processor technically remains in its pre-interrupt Kernel/User mode until the completion of the prolog.

B.2 Cause Codes for Memory Accesses and Data/Address Errors

The core-specific Parameter field may be defined by each core product to encode any information that is appropriate for that core, such as the type of memory accessed and whether it was an address or data error. ARC HS/EV/VPX5 processors fill these bits with zeros, whereas ARC EM inserts suitable values to encode the different types of CCM and data/address errors. See your processor Databook for more information on the core-specific parameter codes.

Cause of EV_MachineCheck Exception	Vector	Cause Code	Parameter
Internal Memory Error on Instruction Fetch	0x03	0x04	Core-specific
Internal Memory Error on Data Access	0x03	0x05	Core-specific
Illegal Overlapping MPU Entries	0x03	0x06	Core-specific
Secure vector table in normal memory	0x03	0x10	Core-specific
NSC jump table not in Secure memory	0x03	0x11	Core-specific

B.3 Exception Handling

B.3.1 ECR User-mode Setting for the ARC EM/EV/VPX5 Family of Cores

The ECR U bit indicates that the user/kernel mode in force at the time of the exception was different from the value of the STATUS32.U bit. This can happen when an exception is raised within an interrupt prolog or epilog where the operating mode may have been changed speculatively so that kernel privileges can apply to memory accesses to the kernel stack, or so that user privileges can apply to memory accesses to the user stack. If an exception occurs when a speculative user/kernel mode is in force, the value captured by ERSTATUS.U still reflects the committed value of STATUS32.U, and is not influenced by the speculative kernel/user mode state. An exception handler can determine the actual kernel/user mode in force at the time of taking an exception by taking the negation of ERSTATUS.U if ECR.U is set to 1.

B.4 Result on Overflow from Integer Division

The DIV, DIVU, REM and REMU instructions specify that their result is implementation dependent when overflow occurs and overflow exceptions are disabled. When overflow occurs, due to division by zero, or when dividing -2^{31} by -1 , there is no meaningful result that can be returned. Therefore, the result of such a division operator should not be relied upon by software.

Table B-1 Implementation Behavior on Division Overflow

Core	Conditions	Core-specific behavior
ARC EM	Overflow and STATUS32.DZ == 0	No update to the destination register
ARC HS/EV/ARC VPX5		Write 0 to the destination register

C

Auxiliary Functions for DSP and XY Extensions

C.1 Auxiliary Function that implements the saturation and rounding functionality

```
// sign extend accumulator on left side and 0 extend on right side
reg [W+2*SHAMT-1:0] shin;
if (signed)
    shin = { {SHAMT{ACC[W-1]}}, ACC, {SHAMT{1'b0}} };
else
    shin = { {SHAMT{1'b0}}, ACC, {SHAMT{1'b0}} };
// logic right shift extended value by SHAMT bits
reg [W+2*SHAMT-1:0] shout;
shout = shin >> SHAMT;
// check for saturation
reg sat; // set if result is saturated
// saturate if any of the bits above MSB is unequal to sign bit
sat = S | (shout[W-1:MSB+1] != {(W-MSB-1){ACC[W-1]}});
// rounding
reg [MSB+1:0] rndout;
if ((!DSP_CTRL.convergent && shout[LSB-1]) ||
    (DSP_CTRL.convergent &&
     (shout[LSB-1] && ((shout[LSB-2:0] != 0) || shout[LSB]))))
    rndout = shout[MSB:0] + (1<< LSB);
else
    rndout = shout[MSB:0];
// check if rounding caused overflow
sat = sat | rndout[MSB+1];
// final result
reg [MSB:0] result;
if (sat)
    if (ACC[W-1])
        result = {1'b1, {MSB{1'b0}}}; // sat to min
    else
        result = {1'b0, {MSB{1'b1}}}; // sat to max
else
    result = rndout[MSB:0];
// mask everthing below rounding LSB
```

```
result[LSB-1:0] = 0;
```

C.2 Data Transformation

C.2.1 Realigns Data from Even and Odd XY Banks

```
// Realign pseudocode, realigns data from even/odd XY banks
// parameters:
// d_w: XY datapath width, 32 [future 64]
// d_w_log2: log2(d_w), if d_w == 32 then 5; if d_w == 64 then 6
bit_vector<d_w>                                // return realigned data
    realign(bit_vector<d_w> even,                // data from even memory
            bit_vector<d_w> odd,                 // data from odd memory
            bit_vector<d_w_log2-2> alsb) { // lsbs of pointer address

    bit_vector<3*d_w> i;
    i[d_w-1:0] = even;
    i[2*d_w-1:d_w] = odd;
    i[3*d_w-1:2*d_w] = even;
    return i>>(8*alsb);
}
```

C.2.2 Data Mask: Masks Unused MSB Bytes

```
// DataMask pseudocode, masks unused MSB bytes
// parameters:
// d_w: XY datapath width, 32 [future 64]
bit_vector<d_w>                                // masked data output
    datamask(bit_vector<d_w> datain,            // input data
            bit_vector<2> vw,                  // vector width, see section 8.3.6
            bit_vector<2> cs,                  // container size, see Table 26
            bit_vector<2> ds) {                // data size, see Table 26

    // Note vw, cs and ds are encoded as:
    // encoding | datasize
    // 0        | 8b
    // 1        | 16b
    // 2        | 32b
    // 3        | 64b
    bit_vector<d_w> r;
    int c = power(2, vw + ds - cs);
    for (int i = 0; i < d_w/8; i++)
        r[8*i+7:8*i] = (i < c) ? datain[8*i+7:8*i] : 0;
    return r;
}
```

C.2.3 Data to Container Expansion Function

```
// Data2Container pseudocode, expand data into containers
// parameters:
// d_w: XY datapath width, 32 [future 64]
bit_vector<d_w>                                // expanded data output
    data2container(bit_vector<d_w> datain,      // input data
            bit_vector<2> cs,                  // container size, see Table 26
            bit_vector<2> ds,                  // data size, see Table 26
```

```

        bit msb,           // MSB align
        bit endian,       // little/big endian
        bit sext) {       // Sign extend

    // Note cs and ds are encoded as:
    // encoding | datasize
    //  0      |    8b
    //  1      |   16b
    //  2      |   32b
    //  3      |   64b
    bit_vector<d_w> r;
    int d; // data counter
    int cd; // difference in container and data size
    r = 0;
    d = 0;
    cd = power(2, cs) - power(2, ds);
    for (int i = 0; i < d_w/8; ) {
        if (msb) {
            // MSB aligned in container so skip LSBs
            i += cd;
        }
        // copy input data into container
        if (endian == big) {
            // bigendian, swap data into container
            neg = datain[7+i*8]; // store sign
            for (int j = power(2, ds)-1; j >= 0; j--) {
                r[8*i+7:8*i] = datain[8*(d+j)+7:8*(d+j)];
                i++;
            }
            d += power(2, ds);
        } else {
            // little endian
            for (int j = 0; j < power(2, ds); j++) {
                r[8*i+7:8*i] = datain[8*d+7:8*d];
                neg = datain[8*d+7]; // store sign
                i++; d++;
            }
        }
        if (sext && !msb && neg) {
            // negative sign extend into MSBs of container
            for (int j = 0; j < cd; j++) {
                r[8*i+7:8*i] = 0xff;
                i++;
            }
        }
    }
    return r;
}

```

C.2.4 ReplicateReverse Function

```

// ReplReverse pseudocode, replicate and reverse container order
// parameters:
//  d_w: XY datapath width, 32 [future 64]

```

```

bit_vector<d_w>                                // reversed/replicated data output
    replreverse(bit_vector<d_w> datain,        // input data
                bit_vector<2> vw,             // vector width, see section 8.3.6
                bit_vector<2> cs,             // container size, see Table 26
                bit repl,                     // Replicate containers
                bit rev) {                   // Reverse container order

    // Note vw and cs are encoded as:
    // encoding | datasize
    //  0      |    8b
    //  1      |   16b
    //  2      |   32b
    //  3      |   64b
    bit_vector<d_w> r;
    int cw; // number of bytes in container
    int v;  // number of bytes in vector
    cw = power(2, cs);
    v = power(2, vw);
    if (repl) {
        // repeatedly copy container 0
        for (i = 0; i < v;) {
            for (j = 0; j < cw; j++, i++) {
                r[8*i+7:8*i] = datain[8*j+7:8*j];
            }
        }
    } else if (rev) {
        c = v-cw;
        // copy container by container
        for (i = 0; i < v;) {
            for (j = 0; j < cw; j++, i++, c++) {
                r[8*i+7:8*i] = datain[8*c+7:8*c];
            }
            c -= 2*cw;
        }
    } else {
        r = datain;
    }
    return r;
}

```

C.2.5 Destination Transformation

```

// Destination transformation, for endian
// parameters:
// d_w: XY datapath width
bit_vector<2*d_w>                                // results can be 64b
    dest2endian(bit_vector<2*d_w> datain,        // input data, 64b
                bit_vector<2> cs,                // container size, see Table 26
                bit endian) {                   // little/big endian switch

    // Note cs is encoded as:
    // encoding | datasize
    //  0      |    8b
    //  1      |   16b
    //  2      |   32b

```

```

//      3      |      64b
bit_vector<d_w> r;
int d; // data counter
r = 0;
d = 0;
    if (endian == big) {
        for (int i = 0; i < d_w/8; ) {
            // swap data into container
            for (int j = power(2, cs)-1; j >= 0; j--) {
                r[8*i+7:8*i] = datain[8*(d+j)+7:8*(d+j)];
                i++;
            }
            d += power(2, cs);
        }
    } else {
        r = datain;
    }
return r;
}

```

C.2.6 Destination Memory Function

```

// Destination memory mapping
// parameters:
//  a_w: CPU address width = 32b
//  d_w: XY datapath width = 32b
void dest2mem(bit_vector<2*d_w> datain,           // input data, 64b
              bit_vector<2> vw,                  // vector width
              bit_vector<a_w> addr,               // destination address
              bit_vector<a_w-3> &addr_even,       // Even memory address output
              bit_vector<a_w-3> &addr_odd,        // Odd memory address output
              bit_vector<d_w> &data_even,        // Even memory write data output
              bit_vector<d_w> &data_odd,         // Odd memory write data output
              bit_vector<d_w/8> &byte_en_even,   // Even memory byte write enable
              bit_vector<d_w/8> &byte_en_odd) { // Odd memory byte write enable
    // Note vw is encoded as:
    // encoding | datasize
    //  0      |    8b
    //  1      |   16b
    //  2      |   32b
    //  3      |   64b
    bit_vector<4*d_w> r; // shifted data
    bit_vector<4*d_w/8> be; // shifted byte enables
    r = datain << (addr[2:0]*8);
    be = (power(2,power(2, vw)) - 1) << addr[2:0];
    data_even = r[31:0] | r[95:64];
    data_odd = r[63:32] | r[127:96];
    byte_en_even = be[3:0] | be[11:8];
    byte_en_odd = be[7:4] | be[15:8];
    addr_even = addr[a_w-1:3] + addr[2]; // take next even address if odd start
    addr_odd = addr[a_w-1:3];
}

```

C.3 Container to Data Packing

```

void apply_modifier(mod_t mod) { // the modifier
uint32 addr;           // current address
uint32 addr_nxt;       // next address
int32  offset;         // signed offset
int32  wrap;           // unsigned wrap boundary, minus 1
int32  mask;           // mask for wrapping
    // retrieve pointer value
    addr = AGU_AUX_AP[mod.ptr_reg];
    // get offset
    if ((mod.opc % 2) == 0) {
        offset = AGU_AUX_OS[mod.offset_reg];
    } else {
        offset = unsigned(mod.offset_imm);
    }
    // get wrap boundary or scale offset
    if (mod.opc == 4) {
        wrap = AGU_AUX_OS[mod.wrap_reg];
    } else if (mod.opc == 5 || mod.opc == 6) {
        wrap = unsigned(power(2, mod.wrap_imm+1))-1;
    } else {
        offset = offset << mod.sc;
    }
    // flip pointer and offset bits if bit-reverse
    if (mod.opc == 2 || mod.opc == 3) {
        addr    = bitrev(addr);
        offset  = bitrev(offset);
    }
    // optionally negate offset
    if (mod.dir)
        offset = -offset;
    addr_nxt = addr + offset;
    if (mod.opc == 2 || mod.opc == 3) {
        // flip back pointer bits if bit-reverse, avoid carry prop into lsbs
        addr_nxt = bitrev(addr_nxt); // bitrev is flipping bits in 32b word
        // create a mask to clear lsbs
        if (dest_op)
            // for destination operands only
            lsbmask = -power(2, mod.vw);
        else if (mod.repl) // these two conditions are for source operands only
            lsbmask = -power(2, mod.ds);
        else
            lsbmask = -power(2, mod.vw-mod.cs+mod.ds);
        addr_nxt &= lsbmask;
    } else if (mod.opc >= 4) {
        // wrap
        mask = power(2, log2up(wrap+1))-1; // log2up is base 2 logarithm rounded up
        if (((addr_nxt & ~mask) != (addr & ~mask)) ||
            ((addr_nxt & mask) > wrap)) {
            // went through the wrap boundary
            if (offset > 0) {
                addr_nxt -= wrap + 1; // went over max
            }
        }
    }
}

```



```

        } else {
            addr_nxt += wrap + 1; // went under min
        }
    }
} // else transparent
// update address with next value
AGU_AUX_AP[mod.ptr_reg] = addr_nxt;
}

```

C.3.1 Destination Transformation for Endian Parameters

```

void apply_modifier(mod_t mod) { // the modifier
uint32 addr; // current address
uint32 addr_nxt; // next address
int32 offset; // signed offset
int32 wrap; // unsigned wrap boundary, minus 1
int32 mask; // mask for wrapping
    // retrieve pointer value
    addr = AGU_AUX_AP[mod.ptr_reg];
    // get offset
    if ((mod.opc % 2) == 0) {
        offset = AGU_AUX_OS[mod.offset_reg];
    } else {
        offset = unsigned(mod.offset_imm);
    }
    // get wrap boundary or scale offset
    if (mod.opc == 4) {
        wrap = AGU_AUX_OS[mod.wrap_reg];
    } else if (mod.opc == 5 || mod.opc == 6) {
        wrap = unsigned(power(2, mod.wrap_imm+1))-1;
    } else {
        offset = offset << mod.sc;
    }
    // flip pointer and offset bits if bit-reverse
    if (mod.opc == 2 || mod.opc == 3) {
        addr = bitrev(addr);
        offset = bitrev(offset);
    }
    // optionally negate offset
    if (mod.dir)
        offset = -offset;
    addr_nxt = addr + offset;
    if (mod.opc == 2 || mod.opc == 3) {
        // flip back pointer bits if bit-reverse, avoid carry prop into lsbs
        addr_nxt = bitrev(addr_nxt); // bitrev is flipping bits in 32b word
        // create a mask to clear lsbs
        if (dest_op)
            // for destination operands only
            lsbmask = -power(2, mod.vw);
        else if (mod.repl) // these two conditions are for source operands only
            lsbmask = -power(2, mod.ds);
        else

```

```

        lsbmask = -power(2, mod.vw-mod.cs+mod.ds);
        addr_nxt &= lsbmask;
    } else if (mod.opc >= 4) {
        // wrap
        mask = power(2, log2up(wrap+1))-1; // log2up is base 2 logarithm rounded up
        if (((addr_nxt & ~mask) != (addr & ~mask)) ||
            ((addr_nxt & mask) > wrap)) {
            // went through the wrap boundary
            if (offset > 0) {
                addr_nxt -= wrap + 1; // went over max
            } else {
                addr_nxt += wrap + 1; // went under min
            }
        }
    } // else transparent
    // update address with next value
    AGU_AUX_AP[mod.ptr_reg] = addr_nxt;
}

```

C.3.2 Destination Transformation for Endian Parameters

```

// Destination transformation, for endian
// parameters:
// d_w: XY datapath width
bit_vector<2*d_w>                                // results can be 64b
            dest2endian(bit_vector<2*d_w> datain, // input data, 64b
                        bit_vector<2>          ds, // data size
                        bit                    endian) { // little/big endian switch

    // Note ds is encoded as:
    // encoding | datasize
    //  0      |    8b
    //  1      |   16b
    //  2      |   32b
    //  3      |   64b
    bit_vector<d_w> r;
    int d; // data counter
    r = 0;
    d = 0;
    if (endian == big) {
        for (int i = 0; i < d_w/8; ) {
            // swap bytes in the data fields
            for (int j = power(2, ds)-1; j >= 0; j--) {
                r[8*i+7:8*i] = datain[8*(d+j)+7:8*(d+j)];
                i++;
            }
            d += power(2, ds);
        }
    } else {
        r = datain;
    }
    return r;
}

```

D

Examples

D.1 Operating Mode Switch

The following example switches the operating mode from kernel mode to user mode.

```
=====
Name       : usermode.c
Copyright  : Synopsys Inc.
Description : This is the kernel/user mode example.

Compiling   : ccac -av2hs -g -O3 -Xlib usermode.c irq_handlers.s vectors.s
linker_cmd.txt -o usermode.out
Debugging   : mdb -av2hs -Xlib -Xtimer0 -Xtimer1 -OKN usermode.out
=====
*/

#include <stdio.h>
#include <stdlib.h>
#include <arc/arc_reg.h>

int set_user_mode = 1;
unsigned int stat32_saved = -1;

_Exception void swi_handler(void)
{
    if(set_user_mode)
    {
        unsigned int estat32 = _lr(ERSTATUS);
        estat32 |= 1 << 7; //user mode
        _sr(estat32, ERSTATUS);
    }
    stat32_saved = _lr(ERSTATUS);
    _sr(_lr(ERET) + 4, ERET); //SWI does not change the return address
}

int main(void)
{

    printf("STAT32/K mode 0x%08x!\n", _lr(STATUS32)); //kernel mode on reset
```

```
_swi();

printf("STAT32/U mode 0x%08x (incorrect)!\n", _lr(STATUS32));
set_user_mode = 0;
_sw_i();

printf("STAT32/U mode 0x%08x (saved from swi)!\n", stat32_saved);

return EXIT_SUCCESS;
}
```

E

ARConnect Commands Details

This chapter describes the details of each ARConnect command and the related `CONNECT_WDATA` and `CONNECT_READBACK` registers formats. If an ARConnect command does not use the `CONNECT_WDATA` and `CONNECT_READBACK` registers or does not require any parameters, the respective sections for that command are marked as N/A (not applicable).

E.1 CMD_CHECK_CORE_ID

Field	Value
CONNECT_CMD[7:0]	0x00

A processor core connected to ARConnect is numbered with a `CORE_ID` ranging from 0 to $N-1$ depending on the number of cores connected to ARConnect. Each core can use the `CMD_CHECK_CORE_ID` command to find out its `CORE_ID`. The `CONNECT_READBACK` register returns the `CORE_ID`.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure E-1 `CONNECT_READBACK` Register Format for `CMD_CHECK_CORE_ID` Command

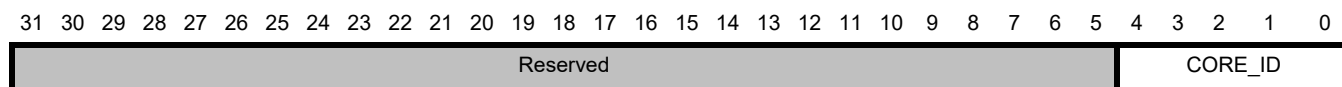


Table E-1 CONNECT_READBACK Register Field Descriptions for CMD_CHECK_CORE_ID Command

Field	Bits	Description
CORE_ID	[4:0]	Indicates core ID of the current core issuing CMD_CHECK_CORE_ID command 0 = CORE0 1 = CORE1 3 = CORE3

E.2 CMD_INTRPT_GENERATE_IRQ

Field	Value
CONNECT_CMD[7:0]	0x01

Each core can be an interrupt initiator core and use the CMD_INTRPT_GENERATE_IRQ command to generate an interrupt to another core or to external system in the system.

The PARAMETER field specified with this command is used to distinguish whether the interrupt is generated to a core or to the external system, and this field specifies the CORE_ID of the interrupt receiver core if the interrupt is generated to the core.

Each core is assigned an internal INTRPT_STATUS register in ICI to record the status of the interrupt (acknowledged or not) from it to all other cores. The CMD_INTRPT_GENERATE_IRQ command sets one bit in the interrupt initiator core's INTRPT_STATUS register corresponding to the interrupt receiver core.

The interrupt line of the interrupt receiver core is asserted after ARConnect issues this command and is asserted until the interrupt receiver core acknowledges all the inter-core interrupts it received.

Parameter Format

The bit [8] of the parameter field defines whether the interrupt is generated to the core or to the external system.

- 1: generate a pulse-sensitive interrupt to external system, and any value in the parameter [7:0] field is ignored.
- 0: generate a level-sensitive interrupt to another core.

The Parameter [4:0] field specifies the CORE_ID of the receiver core. For a four-core system, set its value to 0, 1, 2, or 3.



- Indexing to the core itself is meaningless and is ignored. For example, if the CORE_ID of current core is 3, setting the parameter field equals 3 (Parameter [7:0]=3) accompany with this command is meaningless and no inter-core interrupt is generated.
- In single-core configuration, the ICI can still be used to generate interrupt to external system.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.3 CMD_INTRPT_GENERATE_ACK

Field	Value
CONNECT_CMD[7:0]	0x02

Each core can be an interrupt receiver core and can use the CMD_INTRPT_GENERATE_ACK command to generate an interrupt acknowledgement to the interrupt-initiating core.

The PARAMETER field is used to specify the CORE_ID of the interrupt initiator core.

The CMD_INTRPT_GENERATE_ACK command clears the bit that corresponds to the interrupt receiver core in the interrupt initiator core's INTRPT_STATUS register. However, the interrupt line of this interrupt receiver core is only de-asserted after it serves all the inter-core interrupts it has received.

Parameter Format

The Parameter [4:0] field specifies the CORE_ID of the interrupt initiator core. For a four-core system, set its value to 0, 1, 2, or 3.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.4 CMD_INTRPT_READ_STATUS

Field	Value
CONNECT_CMD[7:0]	0x03

As an interrupt initiator, each core can use the `CMD_INTRPT_READ_STATUS` command to check the interrupt status generated from it to other cores.

The `PARAMETER` field is used to specify the `CORE_ID` of the interrupt receiver core. The `CONNECT_READBACK` register returns the value of the bit corresponding to the interrupt receiver core from the interrupt initiator's `INTRPT_STATUS` register.

Parameter Format

The `Parameter [4:0]` field specifies the `CORE_ID` of the interrupt receiver core. For a four-core system, set its value to 0, 1, 2, or 3.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure E-2 CONNECT_READBACK Register Format for CMD_INTRPT_READ_STATUS Command

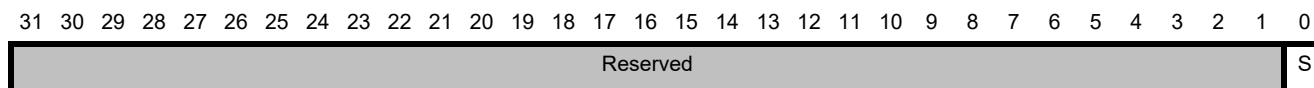


Table E-2 CONNECT_READBACK Register Field Descriptions for CMD_INTERRUPT_READ_STATUS Command

Field	Bits	Description
S	[0]	Interrupt Status of the specified inter-core interrupt 0 = there is no outstanding interrupt or a former interrupt has been serviced (acknowledged) 1 = there is an outstanding interrupt

E.5 CMD_INTRPT_CHECK_SOURCE

Field	Value
CONNECT_CMD[7:0]	0x04

All the inter-core interrupts from other cores to the interrupt receiver core are combined into single interrupt line. An interrupt receiver core can use the `CMD_INTRPT_CHECK_SOURCE` command to find the source of an inter-core interrupt.

The `CONNECT_READBACK` register returns a vector gathered from the internal `INTRPT_STATUS` registers of all interrupt initiator cores.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

For a four-core system, the `CONNECT_READBACK` register returns a vector value after the core issues the `CMD_INTRPT_CHECK_SOURCE` command.

Figure E-3 `CONNECT_READBACK` Register Format for `CMD_INTRPT_CHECK_SOURCE` Command

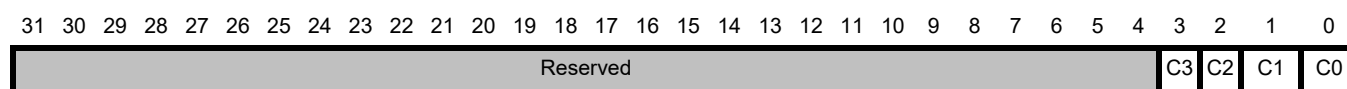


Table E-3 `CONNECT_READBACK` Register Field Descriptions for `CMD_INTRPT_CHECK_SOURCE` Command

Field	Bits	Description
C0	[0]	Indicates whether Core0 is generating an interrupt to the current core. 0 = Core0 is not generating an interrupt to the current core. 1 = Core0 is generating an interrupt to the current core.
C1	[1]	Indicates whether Core1 is generating an interrupt to the current core. 0 = Core1 is generating an interrupt to the current core. 1 = Core1 is generating an interrupt to the current core.
C2	[2]	Indicates whether Core2 is generating an interrupt to the current core. 0 = Core2 is not generating an interrupt to the current core. 1 = Core2 is generating an interrupt to the current core.

Table E-3 CONNECT_READBACK Register Field Descriptions for CMD_INTRPT_CHECK_SOURCE Command

Field	Bits	Description
C3	[3]	Indicates whether Core3 is generating interrupt to the current core. 0 = Core3 is not generating an interrupt to the current core. 1 = Core3 is generating an interrupt to the current core.

E.6 CMD_INTRPT_GENERATE_ACK_BIT_MASK

Field	Value
CONNECT_CMD[7:0]	0x05

Each core can be an interrupt receiver core and can use the CMD_INTRPT_GENERATE_ACK_BIT_MASK command to generate an interrupt acknowledgment to the interrupt-initiating core.



Note

If an external master issues this command to ICI through ASI, the command is treated as a NOP.

The PARAMETER field is used to specify the bit mask of the interrupt initiator core.

The CMD_INTRPT_GENERATE_ACK_BIT_MASK command clears the bit that corresponds to the interrupt receiver core in the interrupt initiator core's INTRPT_STATUS register. However, the interrupt line of this interrupt receiver core is only de-asserted after it serves all the inter-core interrupts it has received.



Note

CMD_INTRPT_GENERATE_ACK_BIT_MASK" can set multi-core bit mask to clear 1 or more initiator core's INTRPT_STATUS register while "CMD_INTRPT_GENERATE_ACK" can only set 1 core.

Interrupt handler can use "CMD_INTRPT_CHECK_SOURCE" to get bit mask information for which cores are set the interrupt to the destination core, if the interrupt handler wants to clear all the initiator cores interrupt status simultaneously, it can use the bit mask result getting from CMD_INTRPT_CHECK_SOURCE to set CMD_INTRPT_GENERATE_ACK_BIT_MASK and clear all the core interrupt at once.

Parameter Format

The parameter [15:0] field specifies the core bit mask of ARC core of the interrupt initiator core, so this commands can be supported in 16 core systems at most.

Each bit[n] is used to indicates each interrupt initiator core whether should be cleared the interrupt INTRPT_STATUS register or not.

- 1: corresponding interrupt initiator core INTRPT_STATUS register is cleared.
- 0: corresponding interrupt initiator core INTRPT_STATUS register is NOT cleared

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.7 CMD_SEMA_CLAIM_AND_READ

Field	Value
CONNECT_CMD[7:0]	0x11

Each core or the external master can use the `CMD_SEMA_CLAIM_AND_READ` command to claim the specified semaphore and verify its ownership. Specify the index of the semaphore in the `PARAMETER` field.

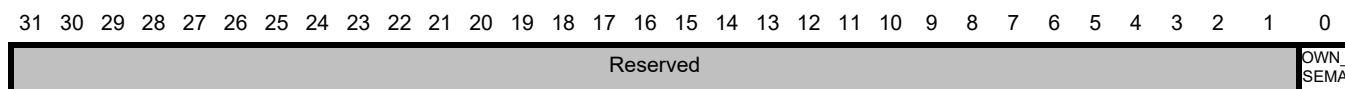
The `CONNECT_READBACK` register returns whether the claim is a success or not.

Parameter Format

The `Parameter [4:0]` field defines the index of the semaphore; Set its value from 0, 1, to N-1 where N is the number of semaphores configured at the system build-time.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format**Figure E-4 CONNECT_READBACK Register Format for CMD_SEMA_CLAIM_AND_READ Command****Table E-4 CONNECT_READBACK Register Field Descriptions for CMD_SEMA_CLAIM_AND_READ Command**

Field	Bits	Description
OWN_SEMA	[0]	Semaphore ownership status 0 = the semaphore is not available to the core, and the claim request has failed. 1 = the core has successfully obtained the semaphore.

E.8 CMD_SEMA_RELEASE

Field	Value
CONNECT_CMD[7:0]	0x12

If the core or the external master has already obtained the ownership of a semaphore, to release the semaphore and allow it to be available for other cores, the core can use the `CMD_SEMA_RELEASE` command.

The `PARAMETER` field specifies the index of the semaphore.

Parameter Format

The `Parameter [4:0]` field defines the index of the semaphore; Set its value from 0, 1, to N-1 where N is the number of semaphores configured in the `CONNECT_SEMA_BUILD` register during the system build-time.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.9 CMD_SEMA_FORCE_RELEASE

Field	Value
CONNECT_CMD[7:0]	0x13

Each core or the external master can use the `CMD_SEMA_FORCE_RELEASE` command to compulsively release and make available a semaphore that has been occupied by other cores in the system.

The `PARAMETER` field specifies the index of the semaphore.

Parameter Format

The `Parameter [4:0]` field defines the index of the semaphore; Set its value from 0, 1, to N-1 where N is the number of semaphores configured in the `CONNECT_SEMA_BUILD` register during the system build-time.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.10 CMD_MSG_SRAM_SET_ADDR

Field	Value
CONNECT_CMD[7:0]	0x21

The message-passing SRAM is an ARConnect local memory. Each core can use the CMD_MSG_SRAM_SET_ADDR command to set the address of message-passing SRAM for subsequent read or write SRAM operations.

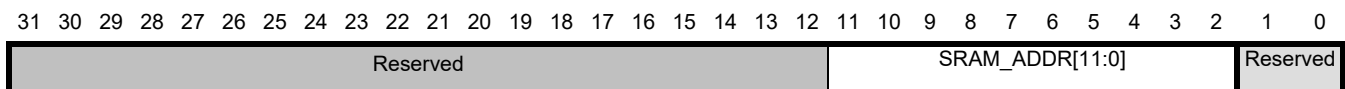
Each core is assigned an internal MSG_ADDR register in ICM to record the address of message-passing SRAM for it. The CONNECT_WDATA register contains the data to be written into the internal MSG_ADDR register.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure E-5 CONNECT_WDATA Register Format for CMD_MSG_SRAM_SET_ADDR Command



Note

Read or write access to the message-passing SRAM is word-aligned, so the two least-significant bits of SRAM_ADDR are always ignored. And, the address exceeding the size of the SRAM field is truncated automatically.

E.11 CMD_MSG_SRAM_READ_ADDR

Field	Value
CONNECT_CMD[7:0]	0x22

Each core or the external master can use the CMD_MSG_SRAM_READ_ADDR command to read the value of its internal MSG_ADDR register.

The CONNECT_READBACK register returns the value of the internal MSG_ADDR register of the core.

Parameter Format

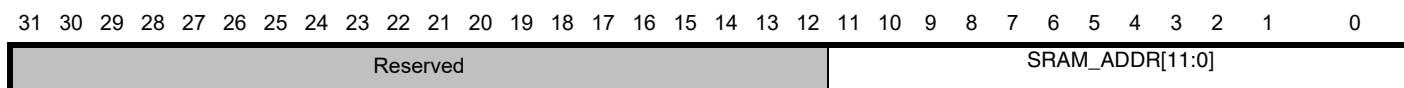
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure E-6 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_ADDR Command



E.12 CMD_MSG_SRAM_SET_ADDR_OFFSET

Field	Value
CONNECT_CMD[7:0]	0x23

Each core or the external master can use the CMD_MSG_SRAM_SET_ADDR_OFFSET command to set the address offset value of the message-passing SRAM for subsequent read or write SRAM operations in incremental mode.

Each core and the external master is assigned an internal MSG_ADDR_OFFSET register in ICM to record the address offset value for the core.

The value of the CONNECT_WDATA register is used with the CMD_MSG_SRAM_SET_ADDR_OFFSET command, to update the internal MSG_ADDR_OFFSET register.

Parameter Format

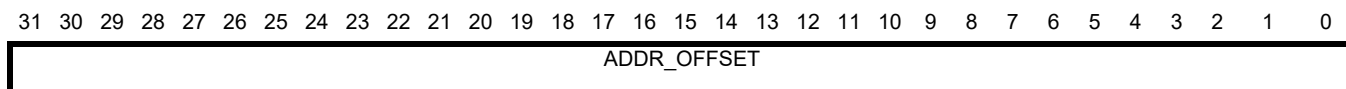
N/A

CONNECT_WDATA Register Format

The offset data must be written into the CONNECT_WDATA register before the CMD_MSG_SRAM_SET_ADDR_OFFSET command is issued as shown in [Figure E-7](#).

The offset value is a signed value and encoded with 2's complement, which ranges from -64 to +60. This value is passed to ICM for subsequent incremental read or write SRAM operations.

Figure E-7 CONNECT_WDATA Register Format for Set-Offset Command



Note

- The offset value must be word aligned, so the two least-significant bits of ADDR_OFFSET are always ignored.
- The hardware result is unpredictable if the written offset value exceeds the allowed range.

CONNECT_READBACK Register Format

N/A

E.13 CMD_MSG_SRAM_READ_ADDR_OFFSET

Field	Value
CONNECT_CMD[7:0]	0x24

Each core or the external master can use the `CMD_MSG_READ_SRAM_ADDR_OFFSET` command to read its address offset value.

The `CONNECT_READBACK` register returns the value of the internal `MSG_ADDR_OFFSET` register of the core.

Parameter Format

N/A

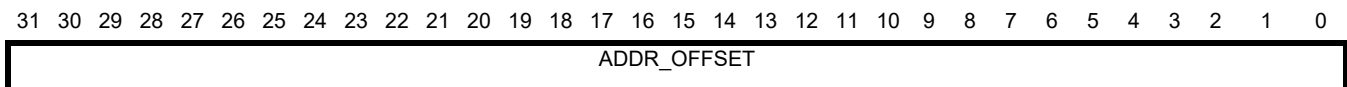
CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the SRAM-address offset value from ICM set by the last `CMD_MSG_SRAM_SET_ADDR_OFFSET` command. The offset value is a signed value and encoded with 2's complement, which ranges from -64 to +60, as shown in [Figure E-8](#).

Figure E-8 `CONNECT_READBACK` Register Format for Read-Offset Command



E.14 CMD_MSG_SRAM_WRITE

Field	Value
CONNECT_CMD[7:0]	0x25

Each core or the external master can use the `CMD_MSG_SRAM_WRITE` command to trigger a write operation to the message-passing SRAM.

The address of message-passing SRAM is determined by the internal `MSG_ADDR` register (which can be updated using the `CMD_MSG_SRAM_SET_ADDR` command, the `CMD_MSG_SRAM_WRITE_INC` command, or the `CMD_MSG_SRAM_READ_INC` command).

The `CONNECT_WDATA` register contains the data to be written into this specified address.

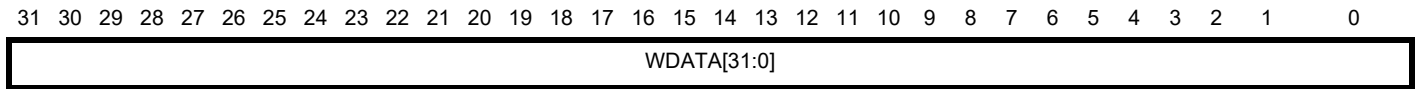
Parameter Format

N/A

CONNECT_WDATA Register Format

The `CONNECT_WDATA` register contains the data to be written into the specified address of SRAM before the write-data commands are sent to the message-passing SRAM.

Figure E-9 `CONNECT_WDATA` Register Format for `CMD_MSG_SRAM_WRITE` Command



CONNECT_READBACK Register Format

N/A

E.15 CMD_MSG_SRAM_WRITE_INC

Field	Value
<code>CONNECT_CMD[7:0]</code>	0x26

To facilitate streamlined write operations to consecutive or regular addresses in the message-passing SRAM, an incremental write mode is supported in ICM.

After setting the address offset value and the start access address, each core can use the `CMD_MSG_SRAM_WRITE_INC` command to continuously write data into the message-passing SRAM without specifying the access address each time.

This command implicitly updates the internal `MSG_ADDR` register by (current `MSG_ADDR` value + `MSG_ADDR_OFFSET` value) after execution.

The `CONNECT_WDATA` register contains the data to be written into current address of message-passing SRAM.

Parameter Format

N/A

CONNECT_WDATA Register Format

The `CONNECT_WDATA` register must contain the data to be written into the specified address of SRAM before write-data commands are sent to the message passing SRAM.

The layout of the `CONNECT_WDATA` is shown in [Figure E-9](#).

CONNECT_READBACK Register Format

N/A

E.16 CMD_MSG_SRAM_WRITE_IMM

Field	Value
CONNECT_CMD[7:0]	0x27

Each core or the external master can use the `CMD_MSG_SRAM_WRITE_IMM` command to write data into the message passing SRAM with the address specified by an immediate value.

The Parameter field specifies this immediate address value. The value must be word aligned, so the two least-significant bits are always ignored.



Note

The `CMD_MSG_SRAM_WRITE_IMM` command does not update the value of core's `MSG_ADDR` register.

Parameter Format

The `Parameter[11:0]` field specifies the SRAM address.

CONNECT_WDATA Register Format

The `CONNECT_WDATA` register must contain the data to be written into the specified address of SRAM before write-data commands are sent to the message passing SRAM.

The layout of the `CONNECT_WDATA` register is shown in [Figure E-9](#).

CONNECT_READBACK Register Format

N/A

E.17 CMD_MSG_SRAM_READ

Field	Value
CONNECT_CMD[7:0]	0x28

Each core or the external master can use `CMD_MSG_SRAM_READ` command to trigger a read operation to the message passing SRAM.

The address of message passing SRAM is determined by the internal `MSG_ADDR` register (which can be updated by the `CMD_MSG_SRAM_SET_ADDR` command, the `CMD_MSG_SRAM_WRITE_INC` command, or the `CMD_MSG_SRAM_READ_INC` command).

The `CONNECT_READBACK` register contains the data read out from this specified address.

Parameter Format

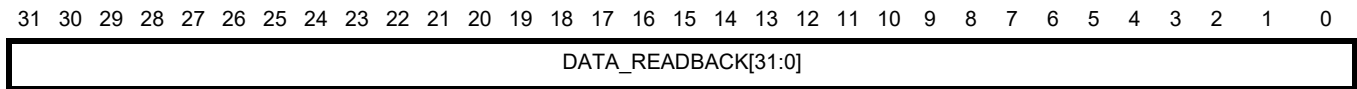
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the data read from specified address of SRAM for the `CMD_MSG_SRAM_READ` command.

Figure E-10 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ Command**E.18 CMD_MSG_SRAM_READ_INC**

Field	Value
CONNECT_CMD[7:0]	0x29

To facilitate streamlined read operations to consecutive or regular addresses in the message-passing SRAM, an incremental read mode is supported in ICM.

After setting the address offset value and the start access address, each core or the external master can use the `CMD_MSG_SRAM_READ_INC` command to continuously read the data from the message-passing SRAM without specifying the access address each time.

This command implicitly updates the internal `MSG_ADDR` register (current `MSG_ADDR` value + `MSG_ADDR_OFFSET` value) after execution.

The `CONNECT_READBACK` register contains the data read out from current address of the message-passing SRAM.

Parameter Format

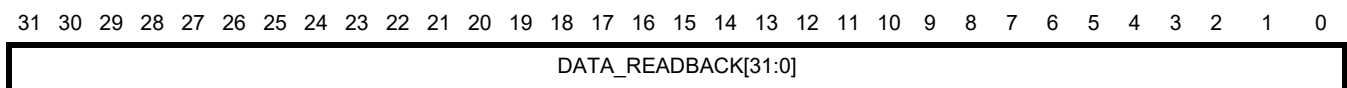
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the data read from the specified address of SRAM for the `CMD_MSG_SRAM_READ_INC` command.

Figure E-11 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_INC Command

E.19 CMD_MSG_SRAM_READ_IMM

Field	Value
CONNECT_CMD[7:0]	0x2A

Each core or the external master can use the `CMD_MSG_SRAM_READ_IMM` command to read the data from the message-passing SRAM using the immediate address. The `PARAMETER` field specifies the immediate address.



Note

The `CMD_MSG_SRAM_READ_IMM` command does not update the core's `MSG_ADDR` register.

Parameter Format

The `Parameter [11:0]` field specifies the SRAM address by an immediate.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the data read from the specified address of SRAM. The layout of `CONNECT_READBACK` register is same as of the `CMD_MSG_SRAM_READ` command.

E.20 CMD_MSG_SET_ECC_CTRL

Field	Value
CONNECT_CMD[7:0]	0x2B

ICM has an internal `MSG_ECC_CTRL` register to enable or disable ECC or parity protection on the Message SRAM. Additionally, flag bits (16 or 17) are updated in the `MSG_ECC_CTRL` register when protection is enabled and non-correctable error occurs.

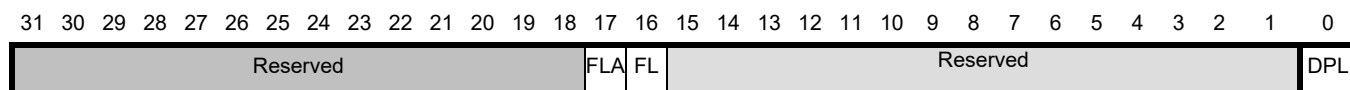
Each core or external master can use the `CMD_MSG_SET_ECC_CTRL` command to set the `MSG_ECC_CTRL` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The data must be written to the `CONNECT_WDATA` register before issuing the `CMD_MSG_SET_ECC_CTRL` command.

Figure E-12 CONNECT_WDATA Register Format for CMD_MSG_SET_ECC_CTRL Command**Table E-5 CONNECT_WDATA Register Field Description for CMD_MSG_SET_ECC_CTRL**

Field	Bits	Description
DPL	[0]	Control bit for ECC/parity protection on the Message SRAM; 0 = enable protection 1 = disable protection On reset, the control bits are set to 0; that is ECC/parity protection is enabled. NOTE: If protection is disabled, the FL flag is never set. When ECC or parity logic is disabled and will never detect and correct errors. NOTE: If ECC is not configured, the value of the DPL bit is always 1, writes are ignored.
FL	[16]	Error Flag 0 : No ECC error detected 1 : Single-bit error (in Parity configuration) or 2-bits errors in data or address (in SECEDED configuration) and 2-bit error in address NOTE: The FL bit can only be set by the ECC hardware logic; it cannot be set by software. But software can use the CMD_MSG_SET_ECC_CTRL command to write 0 to clear this bit. And software can also use the CMD_MSG_READ_ECC_CTRL to read the FL bit. If the software writes 0 and ECC hardware logic also sets this bit simultaneously, the software will preempt.
FLA	[17]	Address single-bit error flag. The FLA bit can only be asserted by the ECC/Parity hardware logic; it cannot be asserted by software. software can use the CMD_MSG_SET_ECC_CTRL command to write 0 to clear this bit, and use the CMD_MSG_READ_ECC_CTRL command to read the FLA value. 0: No error detected 1: Single-bit ECC or parity error in address To clear this flag, the software must write 0 to this bit. If the software writes 0 and ECC hardware logic also sets this bit simultaneously, the software will preempt.

CONNECT_READBACK Register Format

N/A.

E.21 CMD_MSG_READ_ECC_CTRL

Field	Value
CONNECT_CMD[7:0]	0x2C

Each core or external master can use the CMD_MSG_READ_ECC_CTRL command to read the value of the internal MSG_ECC_CTRL register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

This command returns the value of the MSG_ECC_CTRL register. The layout of the CONNECT_READBACK register is same as the CONNECT_WDATA register for the CMD_MSG_SET_ECC_CTRL command.

E.22 CMD_DEBUG_RESET

Field	Value
CONNECT_CMD[7:0]	0x31

Each core or the external master can use the CMD_DEBUG_RESET command to generate reset request to cores in the system. An internal MCD_CORE register in ICD is used to determine the cores in the system that should be reset.

The CONNECT_WDATA register contains the data to be written to the internal MCD_CORE register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The CONNECT_WDATA register must contain the data to be written to the internal MCD_CORE register before issuing the CMD_DEBUG_RESET command.

For a four-core system, the layout of the CONNECT_WDATA register is shown in [Figure E-13](#).

Figure E-13 CONNECT_WDATA Register Format for CMD_DEBUG_RESET Command

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

Figure E-13 CONNECT_WDATA Register Format for CMD_DEBUG_RESET Command**Table E-6 CONNECT_WDATA Field Descriptions for CMD_DEBUG_RESET Command**

Field	Bits	Description
C0	[0]	Core0 select. Indicates whether core0 is selected for the current command 0 = Not selected 1 = Core0 is selected
C1	[1]	Core1select. Indicates whether core1 is selected for the current command 0 = Not selected 1 = Core1 is selected
C2	[2]	Core2 select. Indicates whether core2 is selected for the current command 0 = Not selected 1 = Core2 is selected
C3	[3]	Core3 select. Indicates whether core3 is selected for the current command 0 = Not selected 1 = Core3 is selected

CONNECT_READBACK Register Format

N/A

E.23 CMD_DEBUG_HALT

Field	Value
CONNECT_CMD[7:0]	0x32

Each core or the external master can use the CMD_DEBUG_HALT command to generate halt request to cores in the system. The internal MCD_CORE register in ICD is used to determine the cores in the system that should be halted.

The CONNECT_WDATA register contains the data to be written to the internal MCD_CORE register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

CONNECT_READBACK Register Format

N/A

E.24 CMD_DEBUG_RUN

Field	Value
CONNECT_CMD[7:0]	0x33

Each core or the external master can use the `CMD_DEBUG_RUN` command to generate run request to cores in the system. The internal `MCD_CORE` register in ICD is used to determine the cores in the system that should be run.

The `CONNECT_WDATA` register contains the data to be written into the internal `MCD_CORE` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

CONNECT_READBACK Register Format

N/A

E.25 CMD_DEBUG_SET_MASK

Field	Value
CONNECT_CMD[7:0]	0x34

Each core or the external master can use the `CMD_DEBUG_SET_MASK` command to set the internal `MASK` registers in ICD. The `MASK` register determines whether a global halt is triggered if a core is halted in response to one of the following events: core halt, actionpoint halt, self-halt, and breakpoint halt.

The `PARAMETER` field is used to specify the mask bits.

Each core is assigned an internal `MASK` register. The internal `MCD_CORE` register in the ICD is used to determine the cores in the system for which the `MASK` register should be updated.

The `CONNECT_WDATA` register contains the data to be written into the internal `MCD_CORE` register.

Parameter Format

The `Parameter[3:0]` field specifies the mask bits:

- `Parameter[0]` defines whether a core halt status (whenever the `STATUS32[H]` is set) is masked or not
 - 0: Masked; no global halt triggered when core is halted
 - 1: Not masked; whenever the core is halted, a global halt is triggered
- `Parameter[1]` defines whether an actionpoint caused halt (if actionpoint option is enabled) is masked or not
 - 0: Masked; no global halt triggered when core is halted by an actionpoint
 - 1: Not masked; if an actionpoint caused halt occurs, a global halt is triggered
- `Parameter[2]` defines whether a breakpoint caused halt is masked or not
 - 0: Masked; no global halt triggered when core is halted by a breakpoint
 - 1: Not masked; if a breakpoint caused halt occurs, a global halt is triggered
- `Parameter[3]` defines whether a self-halt is masked or not
 - 0: Masked; no global halt triggered when core is halted by a self-halt
 - 1: Not masked; if a self-halt occurs, a global halt is triggered

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

CONNECT_READBACK Register Format

N/A

E.26 CMD_DEBUG_READ_MASK

Field	Value
<code>CONNECT_CMD[7:0]</code>	0x35

Each core or the external master can use the `CMD_DEBUG_READ_MASK` command to read its internal `MASK` register. The internal `MCD_CORE` register in ICD is used to determine the cores in the system that can `MASK` register.

The `CONNECT_WDATA` register contains the data to be written into the internal `MCD_CORE` register.

The `CONNECT_READBACK` register returns the value of internal `MASK` register of the specified core.

Parameter Format

N/A

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

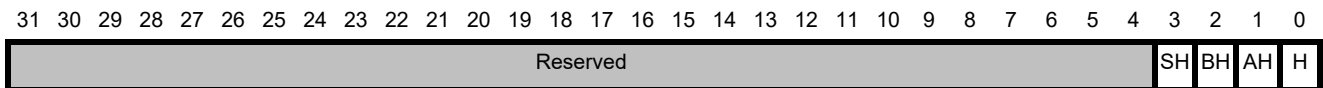


Note

The `CMD_DEBUG_READ_MASK` command supports reading of the internal `MASK` register of one core at a time. That is, only one bit in the `CONNECT_WDATA` can be specified with this command.

CONNECT_READBACK Register Format

Figure E-14 `CONNECT_READBACK` Register Format for `CMD_DEBUG_READ_MASK` Command



The `CONNECT_READBACK` register indicates whether each of the following events trigger a global halt: core halt (H), actionpoint halt (AH), software breakpoint halt (BH), and self halt (SH).



Note

The H bit corresponds to the H field of the `STATUS32` register of the core, and is the complement of the core's EN port.

E.27 CMD_DEBUG_SET_SELECT

Field	Value
<code>CONNECT_CMD[7:0]</code>	0x36

Each core or the external masters can use the `CMD_DEBUG_SET_SELECT` command to select cores that should be halted if the core issuing the command is halted.

The internal `MCD_CORE` register in ICD is used to determine the cores in the system that should be halted.

If a core wants to halt any other cores in the system in response to its own halt events, the core should issue the `CMD_DEBUG_SET_SELECT` command before issuing the `CMD_DEBUG_SET_MASK` command, so that no global halt pulse is skipped.

The `CONNECT_WDATA` register contains the data to be written into the internal `MCD_CORE` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

CONNECT_READBACK Register Format

N/A

E.28 CMD_DEBUG_READ_SELECT

Field	Value
CONNECT_CMD[7:0]	0x37

Each core or the external master can use the `CMD_DEBUG_READ_SELECT` command to read the internal `SELECT` register in ICD.

The `SELECT` register is a common register for all cores, so you need not set any value in the `CONNECT_WDATA` register for core selection.

The `CONNECT_READBACK` register returns the value of internal `SELECT` register

Parameter Format

N/A

CONNECT_WDATA Register Format

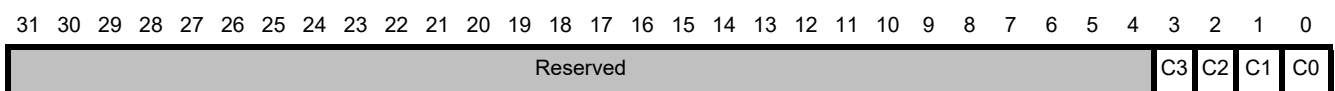
N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the contents of the internal `SELECT` register set by the previous `CMD_DEBUG_SET_SELECT` command.

For a four-core configuration as the example, the register layout is shown in [Figure E-15](#).

Figure E-15 `CONNECT_READBACK` Register Format for `CMD_DEBUG_READ_SELECT` Command



E.29 CMD_DEBUG_READ_EN

Field	Value
CONNECT_CMD[7:0]	0x38

Each core or the external master can use the `CMD_DEBUG_READ_EN` command to read the status, halt or run, of all cores in the system.

The `CONNECT_READBACK` register returns a vector data, each bit corresponding to the status of each core.

Parameter Format

N/A

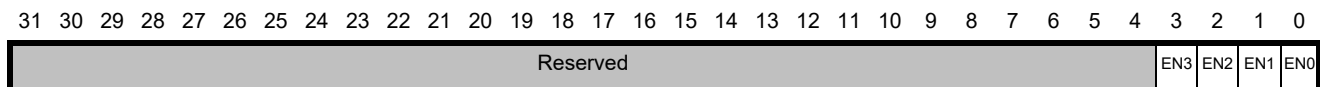
CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CMD_DEBUG_READ_EN` command returns the current state of the EN bits (the complement of the H bit in `STATUS32` register) of all the cores in the system to the `CONNECT_READBACK` register.

For a four-core configuration, the register layout is shown in [Figure E-16](#).

Figure E-16 CONNECT_READBACK Register Format for CMD_DEBUG_READ_EN**Table E-7 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_EN Command**

Field	Bits	Description
EN[n]	[3:0]	The current status of the EN bits ■ 1: indicates that the core is running ■ 0: indicates that the core is halted

E.30 CMD_DEBUG_READ_CMD

Field	Value
CONNECT_CMD[7:0]	0x39

Each core or the external master can use the `CMD_DEBUG_READ_CMD` command to check the last command sent to ICD.

An internal `MCD_CMD` register in ICD records the last command and the parameter data (if needed) it received. The `CONNECT_READBACK` register reads out the value of this internal `MCD_CMD` register.

Parameter Format

N/A

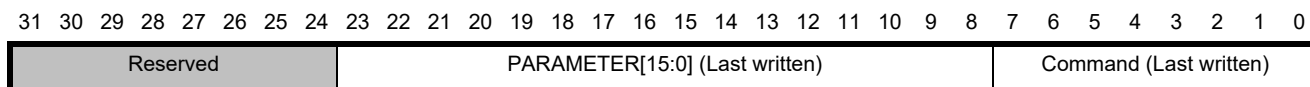
CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CMD_DEBUG_READ_CMD` command returns the last-written contents of the internal `MCD_CMD` register to the `CONNECT_READBACK` register, as shown in [Figure E-17](#).

Figure E-17 `CONNECT_READBACK` Register Format for `CMD_DEBUG_READ_CMD` Command



E.31 `CMD_DEBUG_READ_CORE`

Field	Value
<code>CONNECT_CMD[7:0]</code>	0x3A

Each core or the debugger can use the `CMD_DEBUG_READ_CORE` command to read the value of internal `MCD_CORE` register in ICD.

The `CONNECT_READBACK` register returns the data.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CMD_DEBUG_READ_CORE` command returns the last-written contents of the `MCD_CORE` register to the `CONNECT_READBACK` register. For a four-core configuration, the layout of the `CONNECT_READBACK` register is shown in [Figure E-18](#).

Figure E-18 `CONNECT_READBACK` Register Format for `CMD_DEBUG_READ_CORE` Command

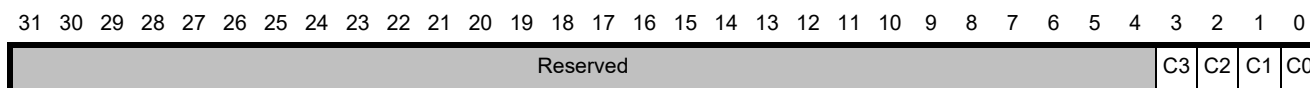


Table E-8 `CONNECT_READBACK` Register Field Descriptions for `CMD_DEBUG_READ_CORE` Command

Field	Bits	Description
C0	[0]	Indicates whether the core0 is selected for the debug-related command 0 = Not selected 1 = Core0 is selected

Table E-8 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_CORE Command

Field	Bits	Description
C1	[1]	Indicates whether the Core1 is selected for the debug-related command 0 = Not selected 1 = Core1 is selected
C2	[2]	Indicates whether the Core2 is selected for the debug-related command 0 = Not selected 1 = Core2 is selected
C3	[3]	Indicates whether the Core3 is selected for the debug-related command 0 = Not selected 1 = Core3 is selected

E.32 CMD_GFRC_CLEAR

Field	Value
CONNECT_CMD[7:0]	0x41

The 64-bit Global Free Running Counter (GFRC) is supported in ARConnect to enable the software to synchronize multiple independent clock domain system.

Each core or the external master can use the CMD_GFRC_CLEAR command to clear GFRC.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.33 CMD_GFRC_READ_LO

Field	Value
CONNECT_CMD[7:0]	0x42

Each core or the external master can use the CMD_GFRC_READ_LO command to read the lower 32-bits of the GFRC counter.

The `CONNECT_READBACK` register returns the lower 32-bit data.

Parameter Format

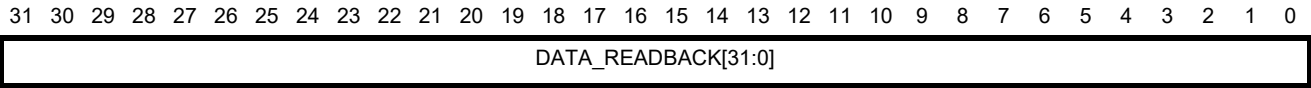
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure E-19 CONNECT_READBACK Register Format for CMD_GFRC_READ_LO Command



E.34 CMD_GFRC_READ_HI

Field	Value
CONNECT_CMD[7:0]	0x43

Each core or the external master can use the `CMD_GFRC_READ_HI` command to read the higher 32-bits of the GFRC counter.

The `CONNECT_READBACK` register returns the higher 32-bit data.

Parameter Format

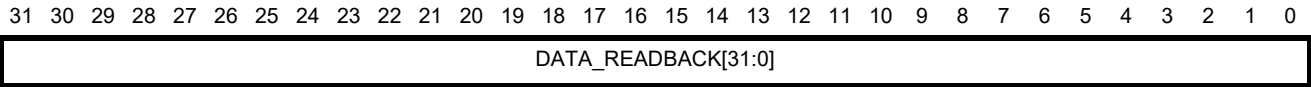
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure E-20 CONNECT_READBACK Register Format for CMD_GFRC_READ_HI Command



E.35 CMD_GFRC_ENABLE

Field	Value
CONNECT_CMD[7:0]	0x44

The GFRC have a DISABLE register to disable or enable all GFRC. Each core or external master can use the CMD_GFRC_ENABLE command to enable GFRC. The DISABLE register is zero after reset, means the GFRC is enabled by default.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.36 CMD_GFRC_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x45

The GFRC have a DISABLE register to disable or enable all GFRC. Each core or external master can use the CMD_GFRC_DISABLE command to disable GFRC. The DISABLE register is zero after reset, means the GFRC is enabled by default.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.37 CMD_GFRC_READ_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x46

Each core or external master can use CMD_GFRC_READ_DISABLE command to read the internal DISABLE register of the GFRC unit.

Parameter Format

N/A

CONNECT_WDATA Register Format

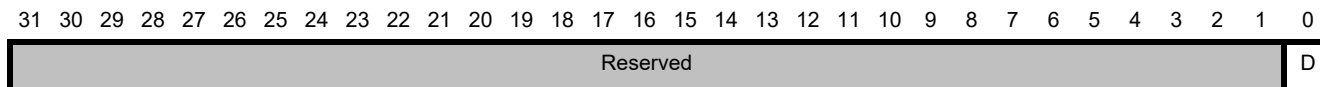
N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the content of internal DISABLE register after CMD_GFRC_READ_DISABLE command.

Figure E-21 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command.

Figure E-21 CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command



E.38 CMD_GFRC_SET_CORE

Field	Value
CONNECT_CMD[7:0]	0x47

The GFRC has an internal CORE register. Each core or the external master can use the CMD_GFRC_SET_CORE command to set the relevant cores to halt the GFRC. The CORE register is zero after reset.

The CONNECT_WDATA register contains the data to be written into the internal CORE register.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure E-21 shows CONNECT_WDATA Register Format for Commands Updating the internal HALT register.

Figure E-22 CONNECT_WDATA Register Format for Command CMD_GFRC_SET_CORE

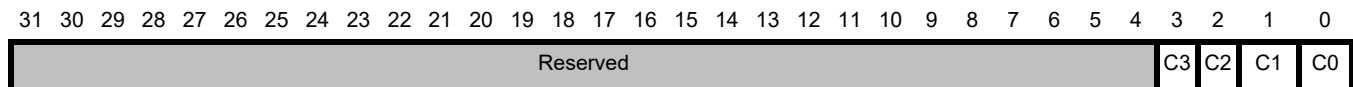


Table E-9 CONNECT_WDATA Field Descriptions

Field	Bits	Description
C0	[0]	CORE0 select Indicate CORE0 is selected for the current command ■ 0 = Not selected ■ 1 = CORE0 is selected
C1	[1]	CORE1 select Indicate CORE1 is selected for the current command ■ 0 = Not selected ■ 1 = CORE1 is selected
C2	[2]	CORE2 select Indicate CORE2 is selected for the current command ■ 0 = Not selected ■ 1 = CORE2 is selected
C3	[3]	CORE3 select Indicate CORE3 is selected for the current command ■ 0 = Not selected ■ 1 = CORE3 is selected

CONNECT_READBACK Register Format

N/A

E.39 CMD_GFRC_READ_CORE

Field	Value
CONNECT_CMD[7:0]	0x48

Each core or the external master can use the CMD_GFRC_READ_CORE command to read the internal CORE register.

The CONNECT_READBACK register returns the value of internal CORE register.

Parameter Format

N/A

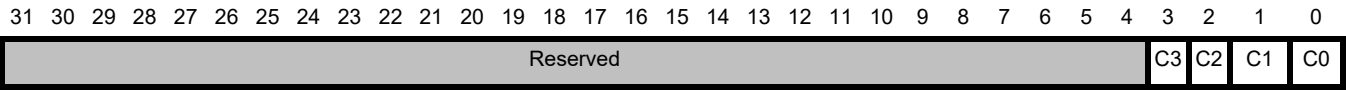
CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal CORE register.

Figure E-23 CONNECT_READBACK Register Format for Command CMD_GFRC_READ_CORE



E.40 CMD_GFRC_READ_HALT

Field	Value
CONNECT_CMD[7:0]	0x49

Each core or the external master can use the CMD_GFRC_READ_HALT command to read the status of GFRC.

The CONNECT_READBACK register returns the value of internal HALT register indicating whether GFRC is halted or not.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

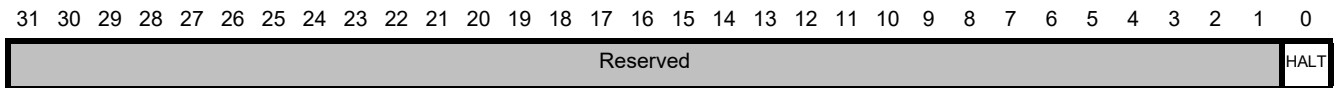
CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal HALT register.

The halt bit is set when all cores specified in the internal CORE register (by CMD_GFRC_SET_CORE command) have been halted. If any one of the core specified in the internal CORE register starts running, the GFRC will start running and the halt bit is cleared.

Figure E-21 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command.

Figure E-24 CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command



E.41 CMD_GFRC_CLK_ENABLE

Field	Value
CONNECT_CMD[7:0]	0x4A

Each core or external master can use the CMD_GFRC_CLK_ENABLE command to enable GFRC clock.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.42 CMD_GFRC_CLK_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x4B

Each core or external master can use the CMD_GFRC_CLK_DISABLE command to disable GFRC clock.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.43 CMD_GFRC_READ_CLK_STATUS

Field	Value
CONNECT_CMD[7:0]	0x4C

Each core or external master can use CMD_GFRC_READ_CLK_STATUS command to read the internal clock status register of the GFRC unit.

Parameter Format

N/A

CONNECT_WDATA Register Format

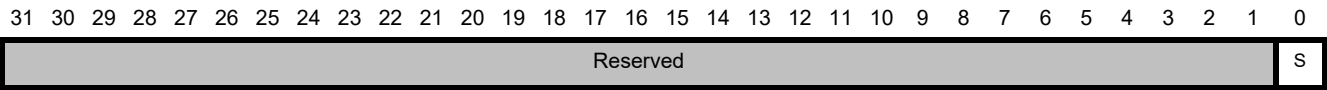
N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the content of internal clock status register after CMD_GFRC_READ_CLK_STATUS command.

Figure E-25 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command.

Figure E-25 CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command



When read result value is 0, it indicates that current GFRC clock is running.

When read result value is 1, it indicates that current GFRC clock is gated off.

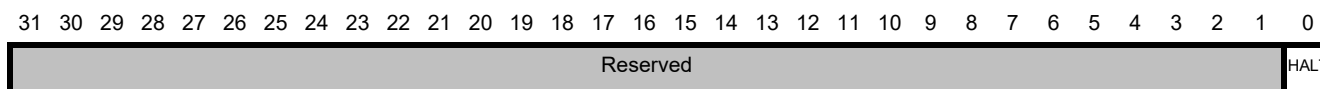
CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal HALT register.

The halt bit is set when all cores specified in the internal CORE register (by CMD_GFRC_SET_CORE command) have been halted. If any one of the core specified in the internal CORE register starts running, the GFRC will start running and the halt bit is cleared.

Figure E-21 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command.

Figure E-26 CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command



E.44 CMD_PDM_SET_PM

Field	Value
CONNECT_CMD[7:0]	0x81

Each core or external master can use the CMD_PDM_SET_PM command to set the power mode for specific ARConnect group.



Note PDM ignores the command to power-down the power group if it has already been in power down mode.

Parameter Format

Parameter indicates group:

- 0x1: to IPC Group

CONNECT_WDATA Register Format

Figure E-27 CONNECT_WDAT Register Format for CMD_PDM_SET_PM Command

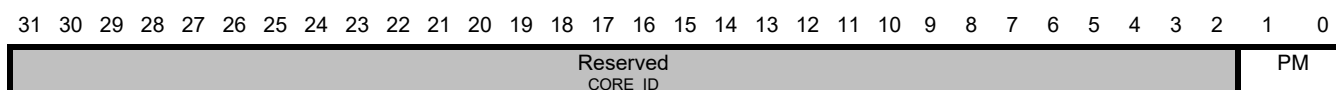


Table E-10 CONNECT_WDATA Register Field Descriptions for CMD_PDM_SET_PM Command

Field	Bits	Description
PM	[1:0]	Indicate power mode 0x0: Power on mode 0x1: Power down mode 1 0x2: Power down mode 2 0x3: Reserved

CONNECT_READBACK Register Format

N/A

E.45 CMD_PDM_READ_PSTATUS

Field	Value
CONNECT_CMD[7:0]	0x82

Each ARConnect group is assigned a PSTATUS register to record its previous power mode, current power mode, power up cause.

Each core or external master can use the CMD_PDM_READ_PSTATUS command to read the internal PM register of the specified ARConnect group.

Parameter Format

Parameter indicates group:

- 0x1: to IPC Group

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns data read out from the PM register of the specified component.

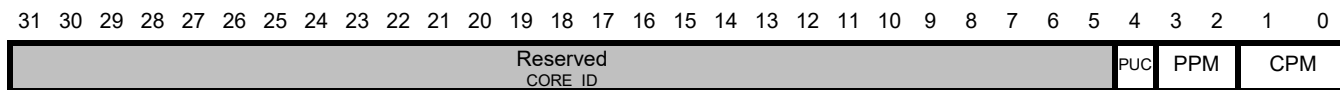
Figure E-28 CONNECT_READBACK Register Format for CMD_PDM_READ_STATUS Command

Table E-11 CONNECT_READBACK Register Field Descriptions for CMD_PDM_READ_STATUS Command

Field	Bits	Description
CPM	[1:0]	Current power mode of IPC group. 0x0: Power on mode 0x1: Power down PM1 mode 0x2: Power down PM2 mode 0x3: Reserved
PPM	[3:2]	Previous power mode of IPC group: 0x0: Power on mode 0x1: Power down PM1 mode 0x2: Power down PM2 mode 0x3: Reserved
PUC	[4]	Power up cause.

E.46 CMD_GFRC_CLEAR

Field	Value
CONNECT_CMD[7:0]	0x41

The 64-bit Global Free Running Counter (GFRC) is supported in ARConnect to enable the software to synchronize multiple independent clock domain system.

Each core or the external master can use the CMD_GFRC_CLEAR command to clear GFRC.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.47 CMD_GFRC_READ_LO

Field	Value
CONNECT_CMD[7:0]	0x42

Each core or the external master can use the CMD_GFRC_READ_LO command to read the lower 32-bits of the GFRC counter.

The CONNECT_READBACK register returns the lower 32-bit data.

Parameter Format

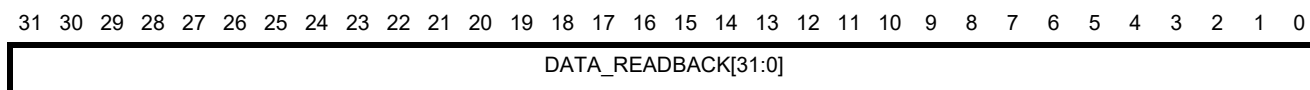
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure E-29 CONNECT_READBACK Register Format for CMD_GFRC_READ_LO Command



E.48 CMD_GFRC_READ_HI

Field	Value
CONNECT_CMD[7:0]	0x43

Each core or the external master can use the CMD_GFRC_READ_HI command to read the higher 32-bits of the GFRC counter.

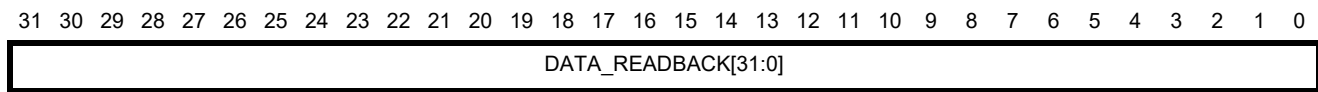
The CONNECT_READBACK register returns the higher 32-bit data.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format**Figure E-30 CONNECT_READBACK Register Format for CMD_GFRC_READ_HI Command****E.49 CMD_GFRC_ENABLE**

Field	Value
CONNECT_CMD[7:0]	0x44

Use the `DISABLE` register to disable or enable GFRC. Each core or external master can use the `CMD_GFRC_ENABLE` command to enable GFRC. The `DISABLE` register is zero after reset: GFRC is enabled by default.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.50 CMD_GFRC_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x45

Use the `DISABLE` register to disable or enable GFRC. Each core or external master can use the `CMD_GFRC_DISABLE` command to disable GFRC. The `DISABLE` register is zero after reset: GFRC is enabled by default.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.51 CMD_GFRC_READ_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x46

Each core or external master can use CMD_GFRC_READ_DISABLE command to read the internal DISABLE register of the GFRC unit.

Parameter Format

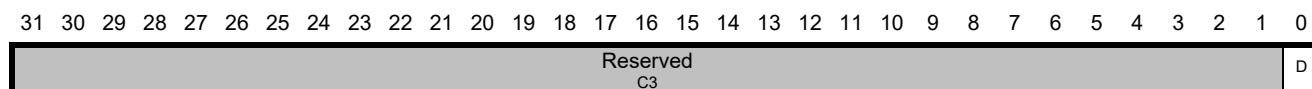
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal DISABLE register after the CMD_GFRC_READ_DISABLE command.

Figure E-31 CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command**E.52 CMD_PMU_SET_PUCNT**

Field	Value
CONNECT_CMD[7:0]	0x51

An internal power-up duration register PUCNT is implemented in PMU to control the applied reset cycles after each power domain exits from the power down mode. Each core or the external master can use the CMD_PMU_SET_PCUNT command to set PUCNT.



If the PUCNT is set to N, reset is applied $(256 \times N) + 8$ cycles after each power domain exits from the power down mode.

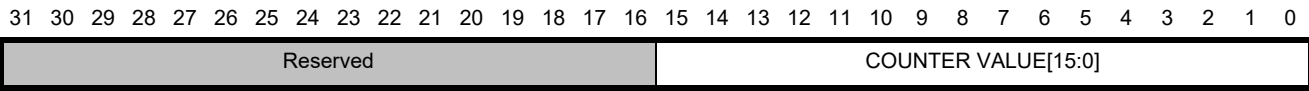
The CONNECT_WDATA register is used to set the value of PUCNT.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure E-32 CONNECT_WDATA Register Format for CMD_PMU_SET_PUCNT Command



CONNECT_READBACK Register Format

N/A

E.53 CMD_PMU_READ_PUCNT

Field	Value
CONNECT_CMD[7:0]	0x52

Each core or the external master can use the CMD_PMU_READ_PUCNT command to read the value of PUCNT. The CONNECT_READBACK register returns the value of PUCNT.

Parameter Format

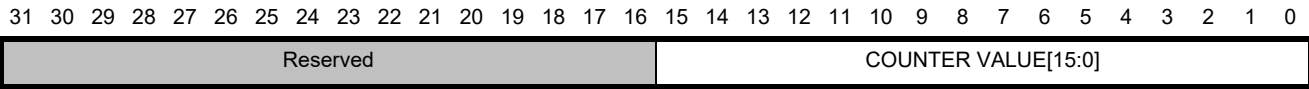
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure E-33 CONNECT_READBACK Register Format for CMD_PMU_READ_PUCNT Command



E.54 CMD_PMU_SET_RSTCNT

Field	Value
CONNECT_CMD[7:0]	0x53

An internal reset delay counter RSTCNT is implemented in the power management unit to extend the reset cycles from isolation de-asserted to reset released. Each core or external master can use the CMD_PMU_SET_RSTCNT command to set the RSTCNT.

**Note**

If the RSTCNT is set to N, the reset duration will extend $(8*N+8)$ cycles after the isolation control is de-asserted.

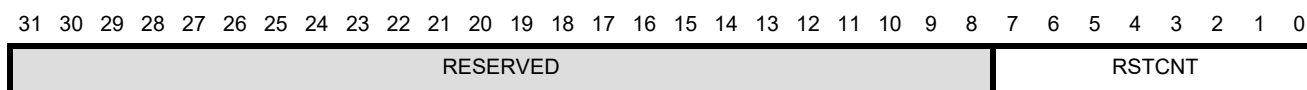
The CONNECT_WDATA register is used to set the value of this reset duration counter.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure E-34 CONNECT_READBACK Register Format for CMD_PMU_SET_RSTCNT Command

**CONNECT_READBACK Register Format**

N/A

E.55 CMD_PMU_READ_RSTCNT

Field	Value
CONNECT_CMD[7:0]	0x54

Each core or external master can use the CMD_PMU_READ_RSTCNT command to read the value of reset duration counter in the PMU unit.

The CONNECT_READBACK register returns the content of reset duration register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The layout of CONNECT_READBACK register is same as the CONNECT_WDATA layout of CMD_PMU_SET_RSTCNT command.

E.56 CMD_PMU_SET_PDCNT

Field	Value
CONNECT_CMD[7:0]	0x55

An internal power-down duration counter PDCNT is implemented in the power management unit to control the cycles from isolation asserted to power switched off. Each core or external master can use the CMD_PMU_SET_PDCNT command to set the PDCNT.



Note

If the PDCNT is set to N, the duration will apply $(256*N+8)$ cycles after the isolation control is asserted. Then the power will be switched off.

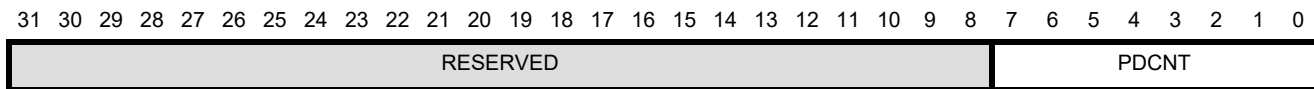
The CONNECT_WDATA register is used to set the value of this power-down duration counter.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure E-35 CONNECT_READBACK Register Format for CMD_PMU_SET_PDCNT Command



CONNECT_READBACK Register Format

N/A

E.57 CMD_PMU_READ_PDCNT

Field	Value
CONNECT_CMD[7:0]	0x56

Each core or external master can use the CMD_PMU_READ_PDCNT command to read the value of power-down duration counter in the PMU unit.

The CONNECT_READBACK register returns the content of power-down duration register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The layout of CONNECT_READBACK register is same as the CONNECT_WDATA layout of CMD_PMU_SET_PDCNT command.

E.58 CMD_PMU_SET_PUCNT

Field	Value
CONNECT_CMD[7:0]	0x51

An internal power-up duration counter PUCNT is implemented in the power management unit to controls the cycles from power switched on to isolation de-asserted. Each core or external master can use the CMD_PMU_SET_PUCNT command to set the PUCNT.



If the PUCNT is set to N, the duration will apply $(256*N+8)$ cycles after each domain is powered up

The CONNECT_WDATA register is used to set the value of this power-up duration counter.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

E.59 CMD_PMU_READ_PUCNT

Field	Value
CONNECT_CMD[7:0]	0x52

Each core or external master can use the CMD_PMU_READ_PUCNT command to read the value of power-up duration counter in the PMU unit.

The CONNECT_READBACK register returns the content of power-up duration register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The layout of CONNECT_READBACK register is same as the CONNECT_WDATA layout of CMD_PMU_SET_PUCNT command.

E.60 Examples: ARConnect ICI

E.60.1 Scenario 1

Core1 generates an inter-core interrupt to Core2. Core2 receives and services the interrupt, and then generates an interrupt acknowledgment to Core1.

In this scenario, Core1 is the interrupt initiator core. To avoid overriding the previous interrupt, Core1 should check the interrupt status before generating a new interrupt to Core2. A segment program is shown in [Example E-1](#).

Example E-1 Segment Program of Scenario 1 — Core1

chk status:	Check the interrupt status of Core2, so that any previous interrupt is not overridden.
SR 0x00000203, [CONNECT_CMD]	Send the CMD_INTRPT_READ_STATUS command to check the status of the interrupt it generated to Core2.
LR r1, [CONNECT_READBACK] CMP r1, 1	The interrupt status is read back using the CONNECT_READBACK register.
BEQ chk_status	Exit the loop until the corresponding status bit is cleared.
SR 0x00000201, [CONNECT_CMD]	Send the CMD_INTRPT_GENERATE_IRQ command to generate inter-core interrupt to Core2.

Core2 is the interrupt receiver core. A program segment of its interrupt handler is shown in [Example E-2](#).

Example E-2 Segment Program of Scenario 1 — Core2

SR r2, [CONNECT_CMD]	The content of r2 is 0x00000004, indicating that the CMD_INTRPT_CHECK_SOURCE command is being sent to check the source of the interrupt.
LR r4, [CONNECT_READBACK]	Read back the vector value from the CONNECT_READBACK register to core register r4.
AND.F 0, r4, 0x00000002 BEQ gen_ack_to_c1	Check whether the bit corresponding to Core1 is asserted.
gen_ack_to_c1: SR r5, [CONNECT_CMD]	The content of r5 is 0x00000102, indicating that the CMD_INTRPT_GENERATE_ACK command is issued to acknowledge the interrupt from Core1. Note: Make sure the interrupt has been de-asserted before exiting the interrupt handler, for example, by using CMD_INTRPT_CHECK_SOURCE command again in a loop until the initiator core has de-asserted the interrupt.

E.60.2 Scenario2

Core0, Core1, and Core2 generate inter-core interrupts to Core3. Core3 services the interrupts and generates interrupt acknowledgments to Core0, Core1, and Core2 respectively.

In this scenario, Core0, Core1, and Core2 are the interrupt initiators. The segment program of each core can be the same as scenario1.

Example E-3 Segment Program of Scenario 2 - Core0, Core1, Core2

chk_status:	Check the interrupt status of Core3, so that any previous interrupt is not overridden.
SR 0x00000303, [CONNECT_CMD]	Send the CMD_INTRPT_READ_STATUS command.
LR r1, [CONNECT_READBACK] CMP.F r1, 0x00000001	The interrupt status is read back from the CONNECT_READBACK register.
BEQ chk_status	Exit the loop until the corresponding status bit is cleared.
SR 0x00000301, [CONNECT_CMD]	Send the CMD_INTRPT_GENERATE_IRQ command to Core3 to generate inter-core interrupt.
...	

Core3 is the interrupt receiver; a segment program of its interrupt handler is shown in [Example E-4](#).

Example E-4 Segment Program of Scenario 2 - Core3

```
serve_irq:
```

```
    SR r0, [CONNECT_CMD]
```

The content of r0 is 0x00000040, which indicates that the CMD_INTRPT_CHECK_SOURCE command is being sent to check the sources of the generated interrupts.

```
    LR r1, [CONNECT_READBACK]
```

The vector value is read back from the CONNECT_READBACK register to core register r1.

```
    CMP r1, 0
```

Check whether all interrupts have been serviced.

```
    BEQ    end_serve_irq
```

```
    ...
```

```
    MOV r2, 1
```

Set the start value 1 for index control.

```
serve_irq_for_each_core:
```

```
    MOV r3, r1
```

Move the interrupt source vector to temporary register r3.

```
    SUB    r10, r2, 1
```

Generate the shift_loop control value.

```
shift_loop:
```

```
    CMP r10, 0
```

```
    BEQ    gen_ack_to_each_core
```

If equal, go to generate interrupt acknowledges.

```
    LSR    r3, r3
```

Shift right operation of interrupt source vector.

```
    SUB    r10, r10, 1
```

```
    BAL    shift_loop
```

```
gen_ack_to_each_core:
```

```
    AND.F 0, r3, 0x00000001
```

Check the value of bit0 for shifted interrupt source vector.

```
    BEQ    incr_index
```

If the AND result is zero, do nothing and increment the index; otherwise, go on, generate interrupt, and acknowledge to the core with a specific index value.

```
    ...
```

```
    SR     r11, [CONNECT_CMD]
```

```
    ...
```

Instructions to generate the command value and store it into r11. In this case, the content of r11 is sent using the command CMD_INTRPT_GENERATE_ACK to Core0, Core1, and Core2 respectively.

```
incr_index:
```

```
    ADD r2, r2, 1
```

```
    CMP r2, CONNECT_CORE_NUMBER
```

BLS serve_irq_for_each_core	If the index value is less than or equal to the number of cores in the system, go on to check the interrupt status.
end_serve_irq:	
RTIE	Note: Ensure that the corresponding bit of the interrupt source has been cleared before exiting the handler.

E.61 Examples: ARConnect Semaphore Unit

E.61.1 Scenario 1

In this scenario, Core1 claims the ownership of semaphore 8. After some processing, Core1 releases semaphore 8.

Example E-5 Segment Program of Scenario 1- Core1

claim: SR 0x00000811, [CONNECT_CMD]	Core1 sends the CMD_SEMA_CLAIM_AND_READ command to claim ownership of semaphore 8.
LR r0, [CONNECT_READBACK]	The claim result is read back from the CONNECT_READBACK register.
CMP r0, 1	Check whether the claim is successful.
BNE claim	If the claim is a failure, go back to claim semaphore 8. Otherwise, continue with the claim.
SR 0x00000812, [CONENCT_CMD]	Send the CMD_SEMA_RELEASE command to release semaphore 8 and allow it to be available for other cores.

E.61.2 Scenario 2

Core2 claims the ownership of semaphore 8, that is already claimed and owned by other cores. Core2 compulsively releases semaphore 8, and then claims its ownership.

Example E-6 Segment Program of Scenario 2 - Core 2

claim: SR 0x00000811, [CONNECT_CMD]	Core2 sends the CMD_SEMA_CLAIM_AND_READ command to claim the ownership of semaphore 8.
LR r0, [CONNECT_READBACK]	The claim result is read from the CONNECT_READBACK to core register r0.
CMP r0, 1	Check whether the claim is successful.
BNE force_release	If the claim is a failure, force release semaphore 8. Otherwise, continue with the claim.

force_release:

```
SR 0x00000813,
[CONNECT_CMD]
```

BA claim

...

Core2 sends the `CMD_SEMA_FORCE_RELEASE` command to compulsively release semaphore 8 and allow it to be available.

E.62 Examples: ICM

E.62.1 Scenario 1

Core1 is the message producer and Core2 is the message consumer. Several messages are to be passed between Core1 and Core2.

Messages are written into and read from the message-passing SRAM. Address 0x00 in the message passing SRAM is used to transfer the communication flag between Core1 and Core2.

Example E-7 Segment Program of Scenario 1 - Core1

SR 0x064, [CONNECT_WDATA]	Set the message passing SRAM address to be 0x064.
SR 0x00000021, [CONNECT_CMD]	Send the <code>CMD_MSG_SRAM_SET_ADDR</code> command to ARConnect.
SR -4, [CONNECT_WDATA]	Set the SRAM address offset value to -4 (a negative value).
SR 0x00000023, [CONNECT_CMD]	Send the <code>CMD_MSG_SRAM_SET_ADDR_OFFSET</code> command.
...	
SR r1 [CONNECT_WDATA]	r1 contains the data to be written into message passing SRAM.
SR 0x00000026, [CONNECT_CMD]	Send the <code>CMD_MSG_SRAM_WRITE_INC</code> command. The data in the <code>CONNECT_WDATA</code> register is written into address 0x064. After that, the SRAM address increments to 0x060 (current address + offset value).
SR r2, [CONNECT_WDATA]	r2 contains the data to be written into the SRAM.
SR 0x00000026, [CONNECT_CMD]	Send the <code>CMD_MSG_SRAM_WRITE_INC</code> command. The data in the <code>CONNECT_WDATA</code> register is written into address 0x060. After that, the SRAM address decrements to 0x05c.
SR 0x55555555, [CONNECT_WDATA]	After producing the message, set address 0x00 to 0x55555555 to tell Core2 that the message is ready.
SR 0x00000027, [CONNECT_CMD]	Send the <code>CMD_MSG_SRAM_WRITE_IMM</code> command. The SRAM address is specified in the parameter field (0x00).

E.63 Examples: ICD

E.63.1 Conditionally Halt Cores Scenario

Core0 is halted by a breakpoint during the execution. The debugger can program ICD to trigger a conditional halt to other cores when a breakpoint halt occurs in Core0.

[Example E-8](#) shows a program segment of Core2 if the debugger programs ICD through Core2.

Example E-8 Conditionally Halt Cores Scenario

```
SR 0x0000000e,  
[CONNECT_WDATA]
```

Core1, Core2, and Core3 are selected for the following debug command.

```
SR 0x00000036,  
[CONNECT_CMD]
```

Send the `CMD_DEBUG_SET_SELECT` command to the ICD, the cores selected by the previous SR instruction are halted by the global halt pulse.

```
SR 0x00000001,  
[CONNECT_WDATA]
```

Core0 is selected for the following debug commands.

```
SR 0x00000434,  
[CONNECT_CMD]
```

Send the `CMD_DEBUG_SET_MASK` command to ICD. Parameter[7:0] is 4'b0100 to indicate the mask value of SH, BH, AH, and H bits to control the generation of the global halt pulse.

E.63.2 Selectively Halt Cores Scenario

Core0 programs ICD to selectively halt other cores in the system.

[Example E-9](#) shows a program segment of Core0.

Example E-9 Selectively Halt Cores Scenario

```
SR 0x0000000e,  
[CONNECT_WDATA]
```

Core1, Core2, and Core3 are selected for the following debug commands.

```
SR 0x00000032,  
[CONNECT_CMD]
```

Send the `CMD_DEBUG_HALT` command to ICD. The cores selected by the previous SR instruction are halted.

E.63.3 Selectively Run Cores Scenario

All cores have already halted. The debugger programs ICD to run all cores in the system.

Example E-10 Selectively Run Cores Scenario

```
SR 0x0000000f,  
[CONNECT_WDATA]
```

Core0 to 3 are selected for following debug command.

```
SR 0x00000033, [CONNECT_CMD]
```

Send the `CMD_DEBUG_RUN` command to ICD. The cores selected by the previous SR instruction start to run.

E.63.4 Selectively Reset Cores Scenario

Core0 is the master CPU of a two core system.

[Example E-11](#) shows a program segment of Core0.

Example E-11 Selectively Reset Cores Scenario

wait_core1_halt:

SR 0x00000038, [CONNECT_CMD]	Send the CMD_DEBUG_READ_EN command to ICD.
LR r0, [CONNECT_READBACK]	
AND.F 0, r0, 0x00000002	Check the EN bit of Core1.
BNZ wait_core1_halt	Reallocate the program of Core1.
SR 0x00000002, [CONNECT_WDATA]	Core1 is selected for the following debug command.
SR 0x00000031, [CONNECT_CMD]	Send the CMD_DEBUG_RESET command to reset Core1.

Example E-12 Segment Program of Scenario 1 - Core2

...	Other tasks are processed then waiting messages are passed from core1.
wait_flag: SR 0x0000002a, [CONNECT_CMD]	Send the CMD_MSG_SRAM_READ_IMM command.
LR r0, [CONNECT_READBACK]	The data is read back from the CONNECT_READBACK register.
CMP r0, 0x55555555	Check whether Core1 produced the message.
BNE wait_flag	If failed, go back to wait for the message.
SR 0x050, [CONNECT_WDATA]	Set the message passing SRAM address to be 0x050.
SR 0x00000021, [CONNECT_CMD]	Send the CMD_MSG_SRAM_SET_ADDR command.
SR 4, [CONNECT_WDATA]	Set the SRAM address offset value to 4 (a positive value).
SR 0x00000023, [CONNECT_CMD]	Send the CMD_MSG_SRAM_SET_ADDR_OFFSET command.
SR 0x00000029, [CONNECT_CMD]	Send the CMD_MSG_SRAM_READ_INC command to read content in address 0x00. The data is read to the CONNECT_READBACK register from the SRAM address 0x050. After that, the address increments to 0x054 (current address + offset value).
LR r1, [CONNECT_READBACK]	Data is read from the address 0x050 is stored to core register r1.
SR 0x00000029, [CONNECT_CMD]	Send the CMD_MSG_SRAM_READ_INC command. The data is read out to the CONNECT_READBACK register from address 0x054. After that, the SRAM address increments to 0x058.

```
LR r2, [CONNECT_READBACK]
```

Data read from the address 0x054 is stored to core register r2.

...

E.64 Examples: GFRC

Example E-13 Segment Program to Read Value of GFRC

```
SR 0x00000042, [CONNECT_CMD]
```

Send the CMD_GRTC_READ_LO command.

```
LR r5, [CONNECT_READBACK]
```

The lower 32-bit value of GRTC are returned to the CONNECT_READBACK register, and stored to r5 register of the core.

```
SR 0x00000043, [CONNECT_CMD]
```

Send the CMD_GRTC_READ_HI command.

```
LR r6, [CONNECT_READBACK]
```

The higher 32-bit value of GRTC is returned to the CONNECT_READBACK register, and store to r6 register of the core.

[Example E-14](#) shows a segment program of the initiator core - Core0.

Example E-14 Segment Program of Common Interrupt 1 Generated by Core 0

<code>_reset:</code>	Reset the exception base
<code> .long _start</code>	
<code> ...</code>	
<code>irq25: .long irq_idu_handler</code>	Common interrupt1 corresponds to IRQ25
<code>test_start:</code>	On-reset start point of the program
<code>.text</code>	
<code>.global _start</code>	
<code>_start:</code>	
<code> ...</code>	
<code>SR 0x00000072, [CONNECT_CMD]</code>	Use the <code>CMD_IDU_DISABLE</code> command to disable the interrupt distribution unit
<code>SR 0x00000011, [CONNECT_WDATA]</code>	Set the distribution mode of common interrupt 1.
<code>SR 0x00000174, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_SET_MODE</code> command to ARConnect, parameter <code>[6:0] = 7'd01</code> indicates common interrupt1 is set
<code>SR 0x0000000f, [CONNECT_WDATA]</code>	Four cores are targeted
<code>SR 0x00000176, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_SET_DEST</code> command to ARConnect, parameter <code>[6:0] = 7'd01</code> indicates that the data in <code>CONNECT_WDATA</code> is written into the <code>DEST</code> register of common interrupt1
<code>SR 0x00000000, [CONNECT_WDATA]</code>	Set the common interrupt MASK to 0
<code>SR 0x0000017C, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_SET_MASK</code> command to ARConnect, parameter <code>[6:0] = 7'd01</code> to indicate that the common interrupt1 will be unmasked.
<code>SR 0x00000071, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_ENABLE</code> command to enable the interrupt distribution unit
<code>SR 0x00000178, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_GEN_CIRQ</code> command to ARConnect, parameter <code>[6:0] = 7'd0</code> to indicate that the common interrupt1 is asserted.

Index

A

- ABS 418
- absolute instruction 418
- actionpoint
 - access requirements 1041
 - auxiliary registers 1039
 - control register 1047
 - extensions to debug register 1032, 1043
 - match mask register 1045
 - registers reset state 1042
 - system 1039
- actionpoint configuration register 1044
- active interrupts register 173
- ADC 420
- ADD 1314
- add instruction
 - syntax summary 317
- add with carry instruction 420
- add with scaled source instruction
 - shift 1 424
 - shift 2 426
 - shift 3 428
- ADD1 424
- ADD2 426
- ADD3 428
- address space 115
- addressing modes 127
 - compare and branch 129
 - conditional branch 129
 - conditional execution 128
 - null instruction format 128
- AE flag 161
- AEX 430
- AND 432
- and instruction 432
- AP_BUILD 1044
- ARCV2
 - introduction 101
 - key features 101
- arithmetic shift left instruction 434
- arithmetic shift right instruction 439
 - multiple 441
- ASL 434
- ASL multiple 436
- ASR 439
- ASR multiple 441
- atomic exchange instruction 544
 - syntax summary 327
- AUX_CACHE_LIMIT 938
- AUX_DCCM 877, 912, 945, 946
- AUX_ICCM 876
- AUX_IRQ_ACT 173, 824, 826
- AUX_IRQ_CTRL 165
- AUX_IRQ_HINT 175
- AUX_RTC_CTRL 1009
- AUX_RTC_HIGH 1012
- AUX_RTC_LOW 1011
- AUX_USER_SP 164, 169, 170, 171, 172
- auxiliary register instructions
 - summary 330
- auxiliary register map 863, 977, 987, 1196, 1201, 1225
- auxiliary register set 863, 911, 944, 977, 987, 1196, 1201, 1225
 - 64-bit rtc 1009
 - actionpoint configuration 1044
 - active interrupts 173
 - barrel shifter configuration 216
 - BCR version 206, 221, 1122, 1127
 - branch target address 200
 - BTA configuration 207, 1124, 1125, 1126, 1129, 1130, 1131, 1132, 1135, 1136, 1137, 1138,

- 1142, 1143, 1148, 1149, 1150, 1151, 1152, 1153, 1156, 1157, 1159, 1161, 1162, 1163, 1165
- build configuration 87, 204, 938, 991, 1173, 1187, 1221, 1299
- core register set configuration 210
- DCCM base address 877, 912, 945, 946
- DCCM RAM configuration 913, 947
- exception cause 186
- exception fault address 198
- exception return address 183
- exception return branch target address 184, 1074
- exception return status 185
- execute indexed base address 182
- ICCM configuration 878
- identity 148
- individual interrupt pending 203
- instruction cache build 909
- instruction fetch queue configuration 883
- interrupt build configuration 221
- interrupt cause 193
- interrupt context saving control 165
- interrupt enable 195
- interrupt P0 status link 163
- interrupt priority 176
- interrupt priority pending 174, 827
- interrupt pulse cancel 202
- interrupt select 194
- interrupt status 197
- interrupt trigger 196
- interrupt vector base 167, 168, 169, 170, 171, 172, 173
- interrupt vector base address configuration 208
- ISA configuration 217
- JLI_BASE 180, 204
- LDI_BASE 181
- loop end 147
- loop start 146
- memory protection unit build configuration 854
- memory protection unit ECR 848
- memory protection unit enable 845
- MIN/MAX configuration 215
- MPU region descriptor base 850
- MPU region descriptor permissions 851
- multiply configuration 212
- non cached memory region 938
- normalize configuration 214
- peripheral memory region 1028
- processor timers 993
- processor timers configuration 1013
- program counter 150
- rtc count high 1012
- rtc count low 1011
- saved user stack pointer 169
- software interrupt 175
- stack region build configuration 870
- status 155
- summary 103
- swap configuration 213
- timer 0 control 997, 1004
- timer 0 count 996, 1003
- timer 0 limit 999, 1005
- timer 1 control 1001, 1007
- timer 1 count 1000, 1006
- timer 1 limit 1002, 1008
- user mode extension enable 198
- auxiliary registers 330
- B**
- B 448
- B_S 450
- barrel shift instructions
 - syntax summary 320
- barrel shifter configuration register 216
- BARREL_BUILD 216
- basecase ALU instructions
 - summary 316, 317, 318
- BBIT0 452
- BBIT1 454
- Bcc 456
- Bcc_S 458
- BCLR 460
- BCR version register 206, 221, 870, 1122, 1127
- BCR_VER 206, 220, 1119
- BH bit 1034
- BI 462
- BIC 464
- BIH 466
- bit clear instruction 460
- bit exclusive-or instruction 496
- bit mask instruction 471, 474
- bit set instruction 482
- bit test instruction 493
- bit toggle instruction 496
- bitwise and with invert instruction 464
- BLcc 468

- BLINK 137
- BMSK 471, 474
- branch and link instruction 468
- branch and link instructions
 - syntax summary 333
- branch conditionally instruction 456
 - 16-bit format 458
- branch indexed 462
- branch indexed to half-word 466
- branch instructions
 - summary 332
 - syntax summary 332
- branch on bit test instruction
 - clear 452
 - syntax summary 333
- branch on bit test set instruction
 - set 454
- branch on compare instruction 476
- branch on compare instructions
 - syntax summary 333
- branch target address register 200
- branch unconditionally instruction 448
- BRcc 476
- breakpoint instruction 479, 782
 - summary 337
 - trap 0 760
- BRK 479
- BSET 482
- BTA 200
- BTA configuration register 207, 1124, 1125, 1126, 1129, 1130, 1131, 1132, 1135, 1136, 1137, 1138, 1142, 1143, 1148, 1149, 1150, 1151, 1152, 1153, 1156, 1157, 1159, 1161, 1162, 1163, 1165
- BTA_LINK_BUILD 207
- BTST 493
- build configuration register
 - actionpoint 1044, 1073
 - barrel shifter 216
 - BCR version 206, 1122, 1127
 - BTA 207, 1124, 1125, 1126, 1129, 1130, 1131, 1132, 1135, 1136, 1137, 1138, 1142, 1143, 1148, 1149, 1150, 1151, 1152, 1153, 1156, 1157, 1159, 1161, 1162, 1163, 1165
 - DCCM RAM 913, 947
 - ICCM 878
 - instruction cache build 909
 - instruction fetch queue 883
 - interrupt 221
 - interrupt vector base address 208
 - ISA 217
 - memory protection unit 854
 - MIN/MAX 215
 - multiply 212
 - normalize 214
 - processor timers 1013
 - stack region 870
- build configuration registers 87, 204, 938, 991, 1173, 1187, 1221, 1299
- BXOR 496
- C**
- canceling pulse triggered interrupts 238
- CLRI 498
- CMP 500
- compare and branch instruction 476
- compare instruction 500
 - reverse compare 691
- condition codes 315
- conditional branch 129
- conditional branch and link instruction 468
- conditional branch instruction 456
 - 16-bit format 458
- conditional execution 128
- conditional jump and link instruction 602
- conditional jump instruction 596
- CONTROL0 997, 1004
- core register map 131, 1175
- core register set 131, 1175
 - extension registers 139, 1176
 - immediate data indicator 139
 - link registers 137
 - loop count 138
 - pointer registers 137
 - reduced configurations 134
 - reserved register r61 139
 - summary 103
 - used in 16-bit instructions 133
- COUNT0 996, 1003
- D**
- data formats 116
 - 16-bit data 124
 - 1-bit data 125
 - 32-bit data 124

- 8-bit data 125
- DCCM_BUILD 913
- DCCM_BUILD configuration register 913
- DEBUG 1032
- debug register 1032
- delayed load from memory instruction 612
- DIV 516, 1371
- DIVU 519, 775, 1302, 1303, 1377, 1380, 1389, 1395, 1405, 1408, 1411, 1413, 1485, 1592, 1597, 1601, 1603, 1650, 1660, 1706, 1708, 1710, 1712, 1716
- DMP_PERIPHERAL 1028
- double fault 287
- DZ bit 160
- E**
- ECR 186
- EI_BASE 182
- EI_S 160, 538
- emulation of extension condition codes 297
- emulation of extension instructions 297
- emulation of extension registers 297
- encoding immediate data 139
- ENTER_S 540
- ERBTA 184
- ERET 183
- ERSTATUS 185
- EX 544
- exception cause register 186
- exception fault address register 198
- exception handling 289
- exception return address register 183
- exception return branch target address register 184, 1074
- exception return status register 185
- exceptions
 - cause codes 265
 - cause register 263, 265
 - data TLB miss 284
 - data TLB protection violation 285
 - delay slots 295
 - double fault 287
 - emulation of extension condition codes 297
 - emulation of extension instructions 297
 - emulation of extension registers 297
 - entry 287
 - exit 291
 - extension instruction 284
 - illegal instruction sequence 282
 - instruction error 281
 - instruction fetch protection violation 279, 285
 - machine check, double fault 278
 - machine check, fatal TLB error 278
 - machine check, instruction fetch memory error 285
 - machine check, overlapping TLB entries 278
 - overview 262
 - parameters 265
 - precision 262
 - priorities 266
 - privilege violation, action point hit on instruction fetch 280
 - privilege violation, actionpoint hit memory or register 287
 - privilege violation, disabled extension 284
 - privilege violation, kernel only access 283
 - privilege violation, misaligned data access 284
 - reset 277
 - types 266
 - vector name 263
 - vector number 264
 - vector offset 264
 - vectors 263
 - with interrupts 287
- exchange instruction 544
 - syntax summary 327
- exclusive-or instruction 777
- EXTB 547
- extend instruction
 - signed byte 728
 - signed half-word 730
 - unsigned byte 547
 - unsigned half-word 549
- extension core registers 139, 1176
- extension instructions
 - summary 338
- extensions
 - auxiliary registers 112
 - condition codes 112
 - core register 112
 - instruction set 112
 - summary 112
- extensions instruction
 - syntax summary 338
- EXTH 549

F

features 101
 FFS 554
 FH bit 780, 1032
 find first set 554
 find last set 561
 FLAG 556, 732
 flag instruction 556, 732
 FLS 561
 force halt 1032
 FP 137
 function epilog sequence 618
 function prolog sequence 540

G

GP 137

H

H bit 162
 halting the processor 780
 host interface 779

I

I_CACHE_BUILD 909
 I_CACHE_BUILD configuration register 909
 ICAUSE 193
 ICCM_BUILD 878
 ICCM_BUILD configuration register 878
 IDENTITY 148
 identity register 148
 IE bit 158
 IFQUEUE_BUILD 883
 IFQUEUE_BUILD configuration register 883
 ILINK 137
 immediate data indicator 139
 instruction error 281
 instruction format
 summary 126
 instruction set
 16 and 32 bit summary 104
 instruction syntax convention 299
 INT_VECTOR_BASE 167
 interrupt
 pending interrupts 235
 interrupt (P0) status link registers 163
 interrupt cause registers 193, 239

interrupt context saving control 165
 interrupt priority pending register 174
 interrupt unit configuration 231
 interrupt unit programming 234
 interrupt vector base address configuration register 208
 interrupt vector base register 167, 168, 169, 170, 171, 172, 173
 interrupts
 canceling pulse triggered interrupts 238
 cause registers 239
 configuration 231
 programming 234
 returning 252
 sensitivity configuration 238
 software triggered 239
 timing 261
 vector base address configuration 234

IRQ_ENABLE 195
 IRQ_PENDING 203
 IRQ_PRIORITY 176
 IRQ_PRIORITY_PENDING 174, 827
 IRQ_PULSE_CANCEL 202
 IRQ_SELECT 194
 IRQ_STATUS 197
 IRQ_TRIGGER 196
 IS bit 781
 ISA configuration register 217
 ISA_CONFIG 217

J

J 593
 Jcc 596
 JL 599
 JLcc 602
 JLI_BASE 178, 179, 180, 204, 1197, 1198
 JLI_S 605
 jump and link conditionally instruction 602
 jump and link indexed 605
 jump and link unconditionally instruction 599
 jump conditionally instruction 596
 jump instructions
 summary 335
 jump unconditionally instruction 593

K

kernel flag instruction [607](#)

kernel mode [224](#)

KFLAG [607](#)

L

L flag [161](#)

LD [612](#)

LDI [616](#)

LDI_BASE [181](#)

LEAVE_S [618](#)

limm [139](#)

link register [240](#)

link registers [137](#)

list of Instructions [409](#), [1301](#)

load from auxiliary register instruction [629](#)
 syntax summary [330](#)

load from memory instruction [612](#)

load pending [780](#)

load register [330](#)

load/store instructions
 summary [325](#)

logical operations [338](#)

logical shift left [633](#)

logical shift left 16 [631](#)

logical shift right [642](#)

logical shift right 16 [640](#)

logical shift right instruction [635](#)
 multiple [637](#)

Long immediate data
 ALU operations [338](#)
 indicator [133](#)

long immediate data
 16-bit instructions [392](#)
 32-bit instructions [307](#), [410](#)

loop count register [138](#)

loop instruction
 syntax summary [335](#)

loop set up instruction [514](#), [622](#)

loops, zero-overhead [297](#)

LP instruction [335](#)

LP_COUNT [138](#), [335](#)

LP_END [147](#), [335](#)

LP_START [146](#), [335](#)

LPcc [514](#), [622](#)

LR [629](#)

LSL16 [631](#)

LSL8 [633](#)

LSR [635](#)

LSR multiple [637](#)

LSR16 [640](#)

LSR8 [642](#)

M

MAX [644](#)

maximum instruction [644](#)

memory based instructions
 summary [325](#)

memory error
 instruction fetch [285](#)

memory prefetch instruction [685](#)

MIN [647](#)

MIN/MAX configuration register [215](#)

minimum instruction [647](#)

MINMAX_BUILD [215](#)

MOV [650](#)

move instruction [650](#), [1468](#), [1472](#)

MPU_BUILD [854](#)

MPU_BUILD configuration register [854](#)

MPU_ECR [848](#)

MPU_EN [817](#), [845](#)

MPY [652](#), [1474](#)

MPYH [655](#), [1494](#)

MPYHU [658](#), [1496](#)

MPYU [661](#), [1498](#)

MPYUW [663](#), [1500](#)

MPYW [665](#), [1504](#)

multiple arithmetic shift right instruction [441](#)

multiply
 high result, signed [1494](#)
 high result, unsigned [658](#), [1496](#)
 signed 16x16 with 32-bit result [665](#), [1504](#)
 unsigned 16x16 with 32-bit result [663](#)

multiply configuration register [212](#)

MULTIPLY_BUILD [212](#)

N

NEG [667](#), [671](#)

negate instruction [667](#), [671](#)

no operation instruction [669](#)

- NOP_S 669
- NORM 673
- NORM_BUILD 214
- normalize configuration register 214
- normalize instruction 673
 - word 675
- NORMW 675
- NOT 678
- not instruction 678
- null instruction format 128
- O**
- operating modes 223
 - summary 104
- operating systems 289
- OR 680
- or instruction 680
- P**
- PC 150
- PCL 139
- plotting results 1110
- pointer registers 137
- pop from stack instruction 683
- POP_S 683
- precise exceptions 262
- PREFETCH 685
- prefetch from memory instruction 685
 - syntax summary 328
- privilege violations 224
- privileged instructions 225
- privileged registers 227
- privileges
 - introduction 223
 - overview 224
- processor timers configuration register 1013
- program counter register 150
- programmer's model 103, 1282
- push onto stack instruction 689
- PUSH_S 689
- R**
- RA bit 1034
- RCMP 691
- register bank 1036
- REM 693, 1561
- REMU 696, 1565
- reserved register r61 139
- reset 277
- return from interrupt/exception instruction 716
 - summary 335
- returning from interrupts 252
- reverse compare instruction 691
- reverse subtract instruction 714
- RF_BUILD 210
- RLC 699
- ROL 701
- ROL8 703
- ROR 705
- ROR multiple 707
- ROR8 710
- rotate left 701
- rotate left through carry instruction 699
- rotate left8 703
- rotate right 8 710
- rotate right instruction 705
 - multiple 707
- rotate right through carry instruction 712
- RRC 712
- RSUB 714
- RTIE 716
- RTOS 289
- S**
- SBC 720
- self halt 1035
- SETcc 722
- SETEQ 723
- SETI 725
- SEXB 728
- SEXH 730
- SH bit 1035
- sign extend instruction
 - byte 728
 - half-word 730
- signed 16x16 multiply
 - with 32-bit result 665, 1504
- signed multiply
 - high result 655, 1494
- single instruction step 781, 1033
- single operand extension instruction

- syntax summary 339
- single step 781
- SLEEP 734, 772
- sleep instruction 734, 772, 1034
 - summary 337
- SMART_CONTROL 1078
- software interrupt instruction 760
- software interrupt register 175
- software triggered interrupt 239
- SP 137
- SR 742
- ST 744
- STACK_REGION_BUILD 870
- STACK_REGION_BUILD configuration register 870
- starting the processor 781
- status register 155
- status save register 240
- STATUS32 155
- store register 330
- store to auxiliary register instruction 742
 - syntax summary 330
- store to memory instruction 744
 - syntax summary 326
- SUB 747
- SUB1 750
- SUB2 752
- SUB3 754
- subtract instruction 747
 - reverse 714
- subtract with carry instruction 720
- subtract with scaled source instruction
 - shift 1 750
 - shift 2 752
 - shift 3 754
- SWAP 444, 446, 756
- swap byte ordering 758
- swap configuration register 213
- swap instruction 444, 446, 756
- SWAP_BUILD 213
- SWAPE 758
- SWI 760
- SYNC 763
- synchronize instruction 763
 - summary 338

- syntax summary
 - add instruction 317
 - atomic exchange instruction 327
 - auxiliary register instructions 330
 - barrel shift instructions 320
 - branch and link instructions 333
 - branch instructions 332
 - branch on bit test instruction 333
 - branch on compare instructions 333
 - exchange instruction 327
 - extension instructions 338
 - jump instructions 335
 - load from auxiliary register instruction 330
 - load/store instructions 325
 - loop instruction 335
 - memory based instructions 325
 - prefetch from memory instruction 328
 - single operand extension instruction 339
 - store to auxiliary register instruction 330
 - store to memory instruction 326
 - zero operand extension instruction 339
 - zero operand instructions 336

T

- test instruction 768
- timer 0 control register 997, 1004
- timer 0 count register 996, 1003, 1011, 1012
- timer 0 limit register 999, 1005
- timer 1 control register 1001, 1007
- timer 1 count register 1000, 1006
- timer 1 limit register 1002, 1008
- Timer COUNT0 996
- TIMER_BUILD 1013
- trap instruction 766
 - summary 338
 - trap 0 760
- TRAP_S 766
- TRAP0 760
- TST 768
- typographic convention 101

U

- U flag 161
- UB bit 1034
- unconditional branch instruction
 - 16-bit format 450
- unconditional jump and link instruction 599
- UNIMP_S 770

unimplemented instruction [770](#)

unsigned 16x16 multiply
with 32-bit result [663](#)

unsigned multiply
high result [658](#), [1496](#)

user mode [224](#)

user mode extension enable register [198](#)

V

VECBASE_AC_BUILD [208](#), [209](#)

X

XOR [777](#)

Z

zero delay loops [138](#), [335](#)

zero extend instruction
byte [547](#)
half-word [549](#)

zero operand extension instruction
syntax summary [339](#)

zero operand instructions
summary [336](#)
syntax summary [336](#)

zero-overhead loop [297](#)

ZZ bit [1034](#)

