



**Synopsys**

## **DesignWare® ARCv2 ISA**

### **Programmer's Reference Manual for ARC EM Processors**

## Copyright Notice and Proprietary Information

© 2017 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 <http://www.synopsys.com/Company/Pages/Trademarks.aspx>.

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

### 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.  
690 E. Middlefield Road  
Mountain View, CA 94043  
[www.synopsys.com](http://www.synopsys.com)

# Contents

Accessing SolvNet .....	59
Contacting the Synopsys Technical Support Center .....	59
Document Revision History .....	60

## Part 1 ARCv2 Baseline ISA ..... 61

### Chapter 1

Introduction .....	63
1.1 Typographic Conventions .....	63
1.2 Key Features of the ARCv2 ISA .....	63
1.3 Programmer's Model .....	66
1.3.1 Core Register Set .....	66
1.3.2 Auxiliary Register Set .....	66
1.3.3 32-bit Instruction Formats .....	66
1.3.4 16-bit Instruction Formats .....	66
1.3.5 Operating Privileges .....	67
1.4 Configurability .....	67
1.4.1 Programming Model Options .....	69
1.4.2 Processor Component Options .....	70
1.4.3 ISA Options .....	71
1.5 Custom Extensions .....	76

1.5.1 Extension Core Registers .....	76
1.5.2 Extension Auxiliary Registers .....	76
1.5.3 Extension Instructions .....	76
1.5.4 Extension Condition Codes .....	76

### Chapter 2

Data Organization and Addressing .....	79
2.1 The ARCv2 Address Space .....	79
2.2 Data Formats .....	80
2.2.1 Vector Operands .....	80
2.2.2 32-Bit Operand Data Elements .....	82
2.2.3 Expansion of Literals .....	82
2.2.4 Expansion of Literals in Vector Instructions .....	82
2.2.5 Expansion of Literals in Store Double (STD) .....	83
2.3 Floating-Point Data Formats .....	84
2.3.1 Single-Precision Floating-Point Data Formats .....	84
2.3.2 Double-Precision Floating-Point Data Formats .....	84
2.4 Double Precision Floating Point Data Formats .....	85

2.4.1 NaN Formats .....	85
2.5 Data Layout in Memory .....	86
2.5.1 64-Bit Data .....	88
2.5.2 32-Bit Data .....	88
2.5.3 16-Bit Data .....	89
2.5.4 8-Bit Data .....	90
2.5.5 1-Bit Data .....	90
2.6 Instruction Layout in Memory .....	90
2.6.1 Addressing Modes .....	92
2.6.2 Null Instruction Format .....	93
2.6.3 Conditional Execution .....	93
2.6.4 Conditional Branch Instruction .....	93
2.6.5 Compare and Branch Instruction .....	94

## Chapter 3

Core Architectural State Registers .....	95
3.1 Core Register Set .....	95
3.1.1 Core Register Mapping Used in 16-bit Instructions .....	97
3.1.2 Reduced Configuration of Core Registers .....	98
3.1.3 Multiple Register Banks .....	98
3.1.4 Illegal Core Register Usage .....	99
3.1.5 Pointer Registers, GP (r26), FP (r27), SP (r28) .....	99
3.1.6 Link Registers, ILINK (r29), BLINK (r31) .....	100
3.1.7 Long Immediate Data Indicator Register, (r30) .....	100
3.1.8 Loop Count Register, LP_COUNT (r60) .....	100
3.1.9 Reserved Register (r61) .....	101
3.1.10 Immediate Data Indicator, LIMM (r62) .....	101
3.1.11 Word-aligned Program Counter, PCL (r63) .....	101
3.2 Extension Core Registers .....	102
3.2.1 Illegal Extension Core Register Usage .....	103
3.3 Auxiliary Register Set .....	104
3.3.1 Baseline Auxiliary Registers .....	104
3.3.2 Optional Architectural, Exception, and Interrupt State Auxiliary Registers .....	105
3.3.3 Register Access Permissions .....	105
3.3.4 Optional Instruction Set Auxiliary Registers .....	106
3.3.5 User Extension Auxiliary Registers .....	107
3.3.6 Optional Build Configuration Registers .....	108
3.3.7 Loop Start Register, LP_START .....	109
3.3.8 Loop End Register, LP_END .....	110
3.3.9 Core Identity Register, IDENTITY .....	111
3.3.10 Program Counter, PC .....	113
3.3.11 Secure Status Register, SEC_STAT .....	114
3.3.12 Status Register, STATUS32 .....	117
3.3.13 Status Register Priority 0, STATUS32_P0 .....	124
3.3.14 Saved User Stack Pointer, AUX_USER_SP .....	125
3.3.15 Interrupt Context Saving Control Register, AUX_IRQ_CTRL .....	126
3.3.16 Interrupt Vector Base Register, INT_VECTOR_BASE .....	128
3.3.17 Secure Interrupt Vector Base Register, INT_VECTOR_BASE_S .....	129
3.3.18 Saved Normal Kernel Stack Pointer, AUX_KERNEL_SP .....	130

3.3.19 Saved Secure User Stack Pointer, AUX_SEC_U_SP .....	131
3.3.20 Saved Secure Kernel Stack Pointer, AUX_SEC_K_SP .....	132
3.3.21 Saved Shadow Normal Stack Pointer, AUX_NSEC_SP .....	133
3.3.22 Active Interrupts Register, AUX_IRQ_ACT .....	134
3.3.23 Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING .....	135
3.3.24 Software Interrupt Trigger, AUX_IRQ_HINT .....	136
3.3.25 Interrupt Priority Register, IRQ_PRIORITY .....	137
3.3.26 Secure Jump and Link Indexed Top Address, NSC_TABLE_TOP .....	139
3.3.27 Secure Jump and Link Indexed Base Address, NSC_TABLE_BASE .....	140
3.3.28 Jump and Link Indexed Base Address, JLI_BASE .....	141
3.3.29 Load Indexed Base Address, LDI_BASE .....	142
3.3.30 Execute Indexed Base Address, EI_BASE .....	143
3.3.31 Exception Return Address, ERET .....	144
3.3.32 Exception Return Branch Target Address, ERBTA .....	145
3.3.33 Exception Return Status, ERSTATUS .....	146
3.3.34 Exception Cause Register, ECR .....	147
3.3.35 Exception Fault Address, EFA .....	148
3.3.36 Exception Secure Status Register, ERSEC_STAT .....	150
3.3.37 Secure Exception Register, AUX_SEC_EXCEPT .....	152
3.3.38 Interrupt Cause Registers, ICAUSE .....	154
3.3.39 Interrupt Select, IRQ_SELECT .....	155
3.3.40 Interrupt Enable Register, IRQ_ENABLE .....	156
3.3.41 Interrupt Trigger Register, IRQ_TRIGGER .....	157
3.3.42 Interrupt Status Register, IRQ_STATUS .....	158
3.3.43 User Mode Extension Enable Register, XPU .....	159
3.3.44 Exception Fault Address, EFA_EXT .....	161
3.3.45 Branch Target Address, BTA .....	162
3.3.46 Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL .....	164
3.3.47 Interrupt Pending Register, IRQ_PENDING .....	165
3.3.48 User Extension Flags Register, XFLAGS .....	166
<b>3.4 Build Configuration Registers .....</b>	<b>166</b>
3.4.1 Build Configuration Registers Version, BCR_VER .....	168
3.4.2 BTA Configuration Register, BTA_LINK_BUILD .....	169
3.4.3 Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD .....	170
3.4.4 Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD .....	171
3.4.5 Core Register File Configuration Register, RF_BUILD .....	172
3.4.6 Multiplier Configuration Register, MULTIPLY_BUILD .....	174
3.4.7 Swap Instruction Configuration Register, SWAP_BUILD .....	175
3.4.8 Normalize Instruction Configuration Register, NORM_BUILD .....	176
3.4.9 Min/Max Instruction Configuration Register, MINMAX_BUILD .....	177
3.4.10 Barrel Shifter Configuration Register, BARREL_BUILD .....	178
3.4.11 Instruction Set Configuration Register, ISA_CONFIG .....	179
3.4.12 Core Architecture Build Configuration Register, ARCH_CONFIG .....	181
3.4.13 Interrupt Build Configuration Register, IRQ_BUILD .....	182
<b>Chapter 4 .....</b>	<b>183</b>
<b>Interrupts and Exceptions .....</b>	<b>183</b>
4.1 Introduction .....	183
4.2 Privileges and Operating Modes .....	183

4.2.1 Kernel Mode .....	184
4.2.2 User Mode .....	184
4.2.3 Privilege Violations .....	184
4.2.4 Switching Between Operating Modes and Privilege Levels .....	187
4.2.5 Switching Between Operating Modes .....	188
4.2.6 Register Replication .....	189
4.3 Interrupts .....	190
4.3.1 Interrupt Unit Features .....	190
4.3.2 Interrupt Unit Configuration .....	191
4.3.3 Interrupt Unit Programming .....	193
4.3.4 Interrupt Handling .....	198
4.4 Exceptions .....	220
4.4.1 Exception Precision .....	220
4.4.2 Exception Vectors and the Exception Cause Register .....	221
4.4.3 Exception Types and Priorities .....	224
4.4.4 Exception Detection .....	244
4.4.5 Effect of Exceptions and Interrupts on Operating Mode .....	244
4.4.6 Exception Entry .....	244
4.4.7 Exception Exit .....	246
4.4.8 Exceptions and Delay Slots .....	247
4.4.9 Emulation of Extension Instructions .....	249
4.4.10 Emulation of Extension Registers and Condition Codes .....	249

## Chapter 5

Instruction Set Summary .....	251
5.1 Instruction Set Encoding .....	251
5.1.1 Top-level Instruction Formats .....	251
5.1.2 Instruction Set Profiles .....	253
5.1.3 Assignment of Instruction Formats to Major Opcodes .....	254
5.1.4 32-bit Instruction Formats .....	256
5.1.5 16-bit Instruction Formats .....	257
5.1.6 Encoding Notation .....	258
5.1.7 Long Immediate Source Operands and Null Destination Operands .....	259
5.1.8 Condition Code Tests .....	260
5.1.9 Encoding Branch and Jump Delay Slot Modes .....	261
5.1.10 Encoding Static Branch Predictions .....	261
5.1.11 Encoding Load / Store Address Write-back Modes .....	262
5.1.12 Encoding Load / Store Cache Bypass Mode .....	263
5.1.13 Encoding Load / Store Data Sizes .....	263
5.1.14 Encoding Load Data Extension Modes .....	264
5.1.15 Encoding Store Data Source Mode .....	264
5.1.16 Use of Reserved Encodings .....	264
5.1.17 Use of Illegal Encodings .....	265
5.2 Instruction Syntax Conventions .....	265
5.3 Status flags and conditions .....	267
5.3.1 Flag Setting .....	267
5.3.2 Status Flags Notation .....	267
5.4 Arithmetic and Logical Instructions .....	268
5.4.1 Integer Adder Operations .....	268

5.4.2 Move Operations .....	269
5.4.3 Bit-wise Logical Operations .....	270
5.4.4 Bit Operations and Mask Operations .....	271
5.4.5 Shift and Rotate Operations .....	271
5.4.6 Selection Operations .....	273
5.4.7 Byte-swapping Operations .....	273
5.4.8 Bit-scanning Operations .....	273
5.4.9 Relational Comparison Operations .....	274
5.4.10 Integer Multiply, Multiply-accumulate, and Divide Operations .....	274
5.4.11 Dual and Quad Integer Multiply / Accumulate Operations .....	275
5.4.12 Integer Vector Operations .....	275
5.4.13 Special and Synchronizing Instructions .....	276
5.5 Memory Access Instructions .....	277
5.5.1 Load Instructions .....	278
5.5.2 Store Instructions .....	279
5.5.3 Stack Pointer Operations .....	279
5.5.4 Atomic Memory Operations .....	280
5.5.5 Prefetch Instructions .....	281
5.6 Auxiliary Register Operations .....	283
5.6.1 Load from Auxiliary Register .....	283
5.6.2 Store to Auxiliary Register .....	283
5.6.3 Auxiliary Register Exchange .....	283
5.7 Control Flow Instructions .....	284
5.7.1 Branch Instructions .....	285
5.7.2 Jump Instructions .....	288
5.7.3 Zero Overhead Loop Instruction .....	288
5.7.4 Return from Interrupt or Exception Instruction .....	289
5.8 Special Instructions .....	289
5.8.1 Breakpoint Instruction .....	290
5.8.2 Sleep Instruction .....	290
5.8.3 Software Interrupt Instruction .....	290
5.8.4 SETI .....	290
5.8.5 CLRI .....	291
5.8.6 Trap Instruction .....	291
5.8.7 Synchronization Instructions .....	291
5.9 APEX Extension Instructions .....	292
5.9.1 Syntax for Generic Extension Instructions .....	292
5.9.2 Syntax for Single Operand Extension Instructions .....	293
5.9.3 Syntax for Zero Operand Extension Instructions .....	293
5.10 Floating Point Instructions .....	294
5.11 DSP Instructions .....	297
5.11.1 Basic Saturating Arithmetic Operations .....	298
5.11.2 Vector Unpacking Operations .....	299
5.11.3 Vector ALU Operations .....	299
5.11.4 Accumulator Operations .....	300
5.11.5 Vector SIMD 16x16 MAC Operations .....	301
5.11.6 Dual Inner Product 16x16 MAC Operations .....	302
5.11.7 32x32 and 16x16 MAC Operations .....	302
5.11.8 Dual 16x8 MAC Operations .....	304

5.11.9 32x16 MAC Operations .....	304
5.11.10 Complex Operations .....	306
<b>Chapter 6</b>	
32-bit Instruction Formats Reference .....	309
6.1 Branch Format [F32_BR0] .....	310
6.1.1 Branch-Conditional Sub-format [F32_BR0, COND] .....	310
6.1.2 Branch Unconditional Far Sub-format [F32_BR0, UCOND_FAR] .....	310
6.2 BRcc, BBITn and BL Format [F32_BR1] .....	311
6.2.1 BRcc / BBITn Register-register Format [F32_BR1, BCC, REG_REG] .....	311
6.2.2 Register-immediate Format [F32_BR1, BCC, REG_U6] .....	311
6.2.3 Branch-and-link Format [F32_BR1, BL] .....	313
6.3 Load Register with Offset Format [F32_LD_OFFSET] .....	315
6.4 Store Register with Offset Format [F32_ST_OFFSET] .....	316
6.5 General Operations Format [F32_GEN4] .....	317
6.5.1 Operator Format Indicators .....	317
6.5.2 Operand Format Indicators .....	317
6.5.3 Syntax and Encoding of Instructions in General Operations Formats .....	318
6.5.4 Dual-operand Instructions, F32_GEN4 .....	321
6.5.5 Move and Compare Instructions, 0x04, [0x0A – 0x0D] and 0x04, [0x11] .....	323
6.5.6 Jump and Jump-and-Link Conditionally, 0x04, [0x20 – 0x23] .....	323
6.5.7 Load Register-Register, 0x04, [0x30 – 0x37] .....	323
6.5.8 Single Operand Instructions, F32_GEN4 .....	324
6.5.9 Zero Operand Instructions, F32_GEN4 .....	324
6.5.10 Dual-operand Instructions, F32_EXT5 .....	326
6.5.11 Single-operand Instructions, F32_EXT5 .....	326
6.5.12 Zero-operand Instructions, F32_EXT5 .....	327
6.5.13 Dual-operand Instructions, F32_EXT6 .....	327
6.5.14 Single-operand Instructions, F32_EXT6 .....	328
6.5.15 Zero-operand Instructions, F32_EXT6 .....	329
6.6 APEX Extension Instruction Format [F32_APEX] .....	330
6.6.1 Extension ALU Operation, Register with Unsigned 6-bit Immediate .....	330
6.6.2 Extension ALU Operation, Register with Signed 12-bit Immediate .....	330
6.6.3 Extension ALU Operation, Conditional Register .....	330
6.6.4 Extension ALU Operation, Conditional Reg with Unsigned 6-bit Immediate .....	331
6.6.5 FastMath Extension Pack Instructions .....	331
6.6.6 CryptoPack Instructions .....	332
<b>Chapter 7</b>	
16-bit Instruction Formats Reference .....	335
7.1 Compact Move/Load [F16_COMPACT] .....	337
7.2 Compact Load/Add/Sub [F16_LD_ADD_SUB] .....	338
7.3 Compact Load/Store [F16_LD_ST_1] .....	339
7.4 Indexed Jump or Execute [F16_JLI_EI] .....	340
7.5 Load / Add Register-Register [F16_LD_ADD_RR] .....	341
7.6 Add/Sub/Shift Register-Immediate [F16_ADD_IMM] .....	342
7.7 Dual Register Operations [F16_OP_HREG] .....	343
7.7.1 Long Immediate Operands in Format 0x0E .....	344
7.8 General Register Format Instructions [F16_GEN_OP] .....	345

7.8.1 DOP-format General Operations .....	345
7.8.2 SOP-format 16-bit General Operations .....	346
7.8.3 ZOP-format 16-bit General Operations .....	347
7.9 16-bit Load and Store Formats with Offset .....	349
7.10 Shift/Subtract/Bit Immediate [F16_SH_SUB_BIT] .....	350
7.11 Stack-based Operations [F16_SP_OPS] .....	351
7.12 Load/Add GP-Relative [F16_GP_LD_ADD] .....	352
7.13 Load PCL-Relative [F16_PCL_LD] .....	353
7.14 Move Immediate [F16_MV_IMM] .....	354
7.15 ADD/CMP Immediate [F16_OP_IMM] .....	355
7.16 Branch on Compare Register with Zero [F16_BCC_REG] .....	356
7.17 Branch Conditionally [F16_BCC] .....	357
7.17.1 Branch Conditionally with cc Field, 0x1E, [0x03, 0x00 – 0x07] .....	357
7.18 Branch and Link Unconditionally [F16_BL] .....	359

## Chapter 8

Instruction Set Details .....	361
8.1 Instruction List .....	362
8.2 ARC Instructions .....	369
ABS .....	370
ADC .....	372
ADD .....	374
ADD1 .....	376
ADD2 .....	378
ADD3 .....	380
AEX .....	382
AND .....	384
ASL .....	386
ASL Multiple .....	388
ASR .....	391
ASR multiple .....	393
ASRL Multiple .....	396
ASR16 .....	399
ASR8 .....	401
B .....	403
B_S .....	405
BBIT0 .....	407
BBIT1 .....	409
Bcc .....	411
Bcc_S .....	413
BCLR .....	415
BI .....	417
BIC .....	419
BIH .....	421
BLcc .....	423
BMSK .....	426
BMSKN .....	429
BRcc .....	432
BRK .....	436

BSET .....	439
BSPEEK .....	441
BSPOP .....	444
BSPUSH .....	447
BTST .....	450
BXOR .....	453
CLRI .....	455
CMP .....	457
DADDH11 .....	459
DADDH12 .....	461
DADDH21 .....	463
DADDH22 .....	465
DEXCL1 .....	467
DEXCL2 .....	469
DBNZ .....	471
DIV .....	473
DIVU .....	476
DMACH .....	479
DMACHU .....	482
DMPYHU .....	488
DMB .....	493
DMULH11 .....	495
DMULH12 .....	497
DMULH21 .....	499
DMULH22 .....	501
DSUBH11 .....	503
DSUBH12 .....	505
DSUBH21 .....	507
DSUBH22 .....	509
EI_S .....	511
ENTER_S .....	513
EX .....	517
EXTB .....	520
EXTH .....	522
FCVT32 .....	524
FCVT32_64 .....	527
FCVT64 .....	530
FCVT64_32 .....	533
FDABS .....	536
FDADD .....	538
FDCMP .....	541
FDCMPF .....	544
FDDIV .....	547
FDMADD .....	550
FDMSUB .....	553
FDMUL .....	556
FDNEG .....	559
FDSQRT .....	561
FDSUB .....	563

FFS .....	566
FLAG .....	568
FLS .....	573
FSABS .....	575
FSADD .....	577
FSCMP .....	580
FSCMPF .....	583
FSDIV .....	586
FSMADD .....	589
FSMSUB .....	592
FSMUL .....	595
FSNEG .....	598
FSSQRT .....	600
FSSUB .....	602
J .....	605
Jcc .....	608
JL .....	611
JLcc .....	614
JLI_S .....	617
KFLAG .....	619
LD LDH LDW LDB LDD LDL .....	624
LDI, LDI_S .....	629
LEAVE_S .....	631
LLOCK .....	635
LLOCKD .....	637
LPcc .....	639
LR .....	646
LSL16 .....	648
LSL8 .....	650
LSR .....	652
LSR multiple .....	654
LSR16 .....	657
LSR8 .....	659
MAC .....	661
MACD .....	664
MACDU .....	667
MACU .....	670
MAX .....	673
MIN .....	676
MOV .....	679
MPY, MPY_S .....	681
MPYD .....	684
MPYDU .....	687
MPYM MPYH .....	690
MPYMU MPYHU .....	693
MPYU .....	696
MPYW, MPYW_S .....	700
NEG .....	702
NOP .....	704

NORM .....	706
NORMH NORMW .....	708
NOT .....	711
OR .....	713
POP .....	715
POP_S .....	716
PREALLOC .....	718
PREFETCH .....	720
PREFETCHW .....	723
PUSH .....	725
PUSH_S .....	726
REM .....	731
REMU .....	734
RLC .....	737
ROL .....	739
ROL8 .....	741
ROR .....	743
ROR multiple .....	745
ROR8 .....	748
RRC .....	750
RSUB .....	752
RTIE .....	754
SBC .....	758
SCOND .....	760
SCONDD .....	762
SETcc .....	764
SETI .....	767
SEXB .....	770
SEXH SEXW .....	772
SFLAG .....	774
SLEEP .....	776
SJLI .....	782
SR .....	784
ST STH STB STD .....	786
SUB .....	790
SUB1 .....	793
SUB2 .....	795
SUB3 .....	797
SWAP .....	799
SWAPE .....	801
SWI .....	803
SYNC .....	806
TRAP_S .....	809
TST .....	811
UNIMP_S .....	813
VADD2H .....	815
VMAC2H .....	821
VMAC2HU .....	824
VMPY2H .....	827

VMPY2HU .....	830
VSUB2H .....	833
WEVT .....	839
WLFC .....	842
XBFU .....	844
XOR .....	846
<b>Chapter 9</b>	
The Host .....	849
9.1 The Host Interface .....	849
9.2 Core and Auxiliary Registers .....	850
9.3 Memory .....	850
9.4 Halting .....	850
9.5 Starting .....	851
9.6 Single Instruction Stepping .....	851
9.6.1 SLEEP Instruction in Single Instruction Step Mode .....	852
9.6.2 BRK Instruction in Single Instruction Step Mode .....	852
9.7 Software Breakpoints .....	852
<b>Part 2</b>	
<b>Security Features .....</b>	<b>853</b>
<b>Chapter 10</b>	
Security Feature Overview .....	855
<b>Chapter 11</b>	
SecureShield™ Technology .....	857
11.1 Features of SecureShield™ Technology .....	857
11.1.1 SecureShield™ Technology Programming Model .....	857
11.1.2 SecureShield Protection Concepts .....	858
11.1.3 SecureShield Modes .....	859
11.1.4 Register Replication .....	860
11.1.5 Configuring SecureShield .....	862
<b>Chapter 12</b>	
Secure Pipeline .....	863
12.1 Secure Build Configuration Register, SEC_BUILD .....	864
<b>Chapter 13</b>	
Secure Memory Protection Unit .....	867
13.1 Protection Overview .....	867
13.2 Configuring an MPU .....	867
13.2.1 Enabling the Memory-Protection Unit .....	868
13.3 Programming MPU Protection .....	868
13.3.1 Programming Considerations .....	869
13.4 Software Lookup of the MPU Table .....	869
13.5 Protection Mechanism .....	870
13.5.1 Protection Exception Priority .....	870
13.6 MPU Auxiliary Register Interface .....	870
13.6.1 MPU Enable Register, MPU_EN .....	872
13.6.2 MPU Probe Register, MPU_PROBE .....	879

13.6.3 MPU Build Configuration Register, MPU_BUILD .....	880
<b>Chapter 14</b>	
Secure Debug Access .....	883
14.1 Secure Debug Access Overview .....	883
14.2 Unlocking a Debug Port .....	883
14.3 Secure Debug Unlock Module Auxiliary Register Interface .....	884
14.3.1 Key Data Register, AUX_KEY_DATA .....	885
14.3.2 Key Unlock Register, AUX_KEY_UNLOCK .....	886
<b>Chapter 15</b>	
Error Protection .....	887
15.1 Error Protection Hardware Control Register, ERP_CTRL .....	888
15.1.1 Register File Error Protection Status Register, RFERP_STATUS_0 .....	891
15.1.2 Register File Error Protection Status Register, RFERP_STATUS_1 .....	893
15.1.3 ECC Syndrome Register, EYSN .....	894
15.2 Error Protection Configuration Register, ERP_BUILD .....	895
15.2.1 .Exceptions .....	897
<b>Part 3</b>	
<b>Memory and System Components .....</b>	<b>899</b>
<b>Chapter 16</b>	
Memory Protection Unit .....	901
16.1 Memory Protection Unit .....	901
16.1.1 Key Features .....	902
16.1.2 Configuration Options .....	902
16.1.3 Enabling the MPU .....	902
16.1.4 Data Organization and Addressing .....	902
16.1.5 Modes of Operation .....	908
16.1.6 Memory Protection Unit Registers .....	911
16.1.7 MPU Enable Register, MPU_EN .....	913
16.1.8 MPU Exception Cause Register, MPU_ECR .....	916
16.1.9 MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB15 .....	918
16.1.10 MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP15 .....	919
16.1.11 Memory Protection Build Configuration Register, MPU_BUILD .....	921
16.1.12 MPU Exceptions .....	922
<b>Chapter 17</b>	
Protection Schemes .....	923
17.1 Stack Checking .....	923
17.1.1 Build Configuration Register .....	924
17.1.2 Enabling Stack Checking .....	924
17.1.3 Specifying Stack Regions .....	924
17.1.4 Stack-Checking Operation .....	925
17.1.5 Exception Information .....	928
17.1.6 Stack Protection Out-of-Bound Limitations .....	928
17.1.7 Stack Region Auxiliary Register Set .....	928
17.1.8 Stack Checking Exceptions .....	934
17.2 Coexisting Protection Schemes .....	935

17.3 Code Protection .....	936
17.3.1 Code Protection Scheme .....	936
17.3.2 Code Protection Behavior .....	936
17.3.3 Code Protection Register, CPROT_BUILD .....	938
<b>Chapter 18</b>	
ICCM .....	939
18.1 ICCM Introduction .....	939
18.2 ICCM base address, AUX_ICCM .....	940
18.3 ICCM Configuration Register, ICCM_BUILD .....	941
18.4 Exceptions .....	942
<b>Chapter 19</b>	
Instruction Fetch Queue .....	943
19.1 IFQ Introduction .....	943
19.1.1 Instruction Fetch Queue Configuration Register, IFQUEUE_BUILD .....	943
<b>Chapter 20</b>	
Instruction Cache .....	945
20.1 Instruction Cache Auxiliary Registers .....	945
20.2 Auxiliary Registers .....	945
20.2.1 Invalidate Instruction Cache, IC_IVIC .....	947
20.2.2 Instruction Cache Control Register, IC_CTRL .....	948
20.2.3 Lock Instruction Cache Line, IC_LIL .....	949
20.2.4 Invalidate Instruction-Cache Line, IC_IVL .....	951
20.2.5 Instruction Cache External-Access Address, IC_RAM_ADDR .....	952
20.2.6 Instruction-Cache Tag Access, IC_TAG .....	953
20.2.7 Instruction Cache Secure Bit Tag Register, IC_XTAG .....	955
20.2.8 Instruction Cache Data Access, IC_DATA .....	956
20.3 Instruction Cache Configuration Register, I_CACHE_BUILD .....	957
20.4 Exceptions .....	958
<b>Chapter 21</b>	
DCCM .....	959
21.1 DCCM Introduction .....	959
21.2 DCCM Base Address, AUX_DCCM .....	960
21.2.1 DCCM Configuration Register, DCCM_BUILD .....	961
21.2.2 Exceptions .....	962
<b>Chapter 22</b>	
Data Cache .....	963
22.1 Data Cache Auxiliary Registers .....	963
22.1.1 Invalidate Data Cache, DC_IVDC .....	965
22.1.2 Data Cache Control Register, DC_CTRL .....	967
22.1.3 Lock Data Cache Line, DC_LDL .....	968
22.1.4 Invalidate Data Line, DC_IVDL .....	971
22.1.5 Flush Data Cache, DC_FLSH .....	973
22.1.6 Flush Data Line, DC_FLDL .....	975
22.1.7 Data Cache External Access Address, DC_RAM_ADDR .....	977
22.1.8 Data Cache Tag Access, DC_TAG .....	978

22.1.9 Data Cache Secure Bit Register, DC_XTAG .....	980
22.1.10 Data Cache Data Access, DC_DATA .....	981
22.1.11 Non-cached Memory Region, AUX_CACHE_LIMIT .....	982
22.2 Build Configuration Registers .....	983
22.2.1 Data Cache Configuration Register, D_CACHE_BUILD .....	984
22.2.2 Exceptions .....	985
 Chapter 23	
Program Storage .....	987
23.1 NV_ICCM Register Interface .....	987
23.1.1 NVM ICCM Control Register, NV_ICCM_CTRL .....	988
23.1.2 NVM ICCM Erase Register, NV_ICCM_ERASE .....	990
23.1.3 NVM ICCM Programming Register, NV_ICCM_PROG .....	991
23.1.4 NVM ICCM Data Register, NV_ICCM_DATA .....	992
23.2 Exceptions .....	992
 Chapter 24	
Cryptographic Key Storage .....	993
24.1 Cryptographic Key Storage Introduction .....	993
24.2 Cryptographic Key Storage Register Interface .....	993
24.2.1 Key Store Control Register, KEY_STORE_CTRL .....	995
24.2.2 Key Store Password Register, KEY_STORE_PWD .....	997
24.2.3 Key Store Address Register, KEY_STORE_ADDR .....	998
24.2.4 Key Store Data Register, KEY_STORE_DATA .....	999
24.2.5 Key Store SID Register, KEY_STORE_SID .....	1000
24.2.6 Key Store Build Configuration Register, KEY_STORE_BUILD .....	1001
 Chapter 25	
Calibration Parameter Storage .....	1003
25.0.1 Calibration Parameter Store Control Register, CAL_STORE_CTRL .....	1004
25.0.2 Calibration Parameter Store Address Register, CAL_STORE_ADDR .....	1006
25.0.3 Calibration Parameter Store Data Register, CAL_STORE_DATA .....	1007
25.1 Build Configuration Registers .....	1007
25.1.1 Calibration Parameter Store Build Configuration Register, CAL_STORE_BUILD .....	1008
 Chapter 26	
Processor Timers .....	1009
26.1 Timers .....	1009
26.2 Auxiliary Timer Registers .....	1010
26.2.1 Timer 0 Count Register, COUNT0 .....	1012
26.2.2 Timer 0 Control Register, CONTROL0 .....	1013
26.2.3 Timer 0 Limit Register, LIMIT0 .....	1014
26.2.4 Timer 1 Count Register, COUNT1 .....	1015
26.2.5 Timer 1 Control Register, CONTROL1 .....	1016
26.2.6 Timer 1 Limit Register, LIMIT1 .....	1017
26.2.7 Secure Timer 0 Count Register, AUX_ST0_COUNT .....	1018
26.2.8 Secure Timer 0 Control Register, AUX_ST0_CTRL .....	1019
26.2.9 Secure Timer 0 Limit Register, AUX_ST0_LIMIT .....	1020
26.2.10 Secure Timer 1 Count Register, AUX_ST1_COUNT .....	1021
26.2.11 Secure Timer 1 Control Register, AUX_ST1_CTRL .....	1022

26.2.12	Secure Timer 1 Limit Register, AUX_ST1_LIMIT	1023
26.2.13	RTC Control Register, AUX_RTC_CTRL	1024
26.2.14	RTC Count Low Register, AUX_RTC_LOW	1026
26.2.15	RTC Count High Register, AUX_RTC_HIGH	1027
26.2.16	Timers Configuration Register, TIMER_BUILD	1028
26.3	Watchdog Timers	1029
26.4	Watchdog Password Register, WDT_PASSWD	1030
26.5	Watchdog Timer Register, WDT_CTRL	1031
26.6	Watchdog Timer Period Register, WDT_PERIOD	1032
26.7	Watchdog Timer Count Register, WDT_COUNT	1033
<b>Chapter 27</b>		
Safety Island	.....	1035
27.1	Safety Island Overview	1035
27.2	Dual-Core Lockstep Initialization After Reset	1036
27.3	Halting and Running Cores that are in Lockstep	1037
27.4	Safety Island Auxiliary Register Interface	1037
27.4.1	Lockstep Control Register, LSC_CTRL	1038
27.4.2	Lockstep Debug Control Register, LSC_MCD	1040
27.4.3	Lockstep Status Register, LSC_STATUS	1041
27.4.4	Lockstep Error Register, LSC_ERROR_REG	1042
27.4.5	Lockstep Configuration Register, LSC_BUILD_AUX	1044
<b>Chapter 28</b>		
Peripheral Bus	.....	1045
28.1	DMP Auxiliary Registers	1045
28.2	Peripheral Memory Region, DMP_PER_AUX	1046
<b>Chapter 29</b>		
External Host Debugging	.....	1047
29.1	External Host Debugging Introduction	1047
29.2	Host Debug Register Interface	1047
29.3	Debug Register, DEBUG	1048
29.3.1	Interrupt and Register Bank Debug Register, DEBUGI	1052
29.3.2	Architectural Clock Gating Control Register, ACG_CTRL	1054
<b>Chapter 30</b>		
Debug Features	.....	1055
30.1	Debug Introduction	1055
30.2	Actionpoints	1055
30.2.1	Actionpoint Auxiliary Registers	1055
30.2.2	Actionpoint Extensions to Debug Register, DEBUG, 0x05	1059
30.2.3	Action Points and Architectural Clock Gating	1060
30.2.4	Actionpoints Configuration Register, AP_BUILD	1060
30.2.5	Actionpoint Match Value, AP_AMV0	1060
30.2.6	Actionpoint Match Mask, AP_AMM0	1061
30.2.7	Actionpoint Control, AP_AC0	1062
30.2.8	Actionpoint Match Value, AP_AMV1	1066
30.2.9	Actionpoint Match Mask, AP_AMM1	1066
30.2.10	Actionpoint Control, AP_AC1	1067

30.2.11 Actionpoint Match Value, AP_AMV2 .....	1067
30.2.12 Actionpoint Match Mask, AP_AMM2 .....	1067
30.2.13 Actionpoint Control, AP_AC2 .....	1068
30.2.14 Actionpoint Match Value, AP_AMV3 .....	1068
30.2.15 Actionpoint Match Mask, AP_AMM3 .....	1068
30.2.16 Actionpoint Control, AP_AC3 .....	1068
30.2.17 Actionpoint Match Value, AP_AMV4 .....	1069
30.2.18 Actionpoint Match Mask, AP_AMM4 .....	1069
30.2.19 Actionpoint Control, AP_AC4 .....	1069
30.2.20 Actionpoint Match Value, AP_AMV5 .....	1070
30.2.21 Actionpoint Match Mask, AP_AMM5 .....	1071
30.2.22 Actionpoint Control, AP_AC5 .....	1071
30.2.23 Actionpoint Match Value, AP_AMV6 .....	1071
30.2.24 Actionpoint Match Mask, AP_AMM6 .....	1071
30.2.25 Actionpoint Control, AP_AC6 .....	1072
30.2.26 Actionpoint Match Value, AP_AMV7 .....	1073
30.2.27 Actionpoint Match Mask, AP_AMM7 .....	1073
30.2.28 Actionpoint Control, AP_AC7 .....	1073
30.2.29 Watchpoint Program Counter, AP_WP_PC .....	1074
30.3 SmaRT .....	1074
30.3.1 Features .....	1074
30.3.2 Overview of SmaRT Operation .....	1075
30.3.3 Auxiliary Registers .....	1076
30.3.4 SMART_BUILD Configuration Register, SMART_BUILD .....	1076
30.3.5 SmaRT Control Register, SMART_CONTROL .....	1077
30.3.6 Smart Data Register, SMART_DATA .....	1078
30.3.7 SRC_ADDR Value, Index 0 .....	1078
30.3.8 DEST_ADDR Value, Index 1 .....	1079
30.3.9 FLAGS Value, Index 2 .....	1079

## Chapter 31

Performance Counters .....	1081
31.1 Performance Counters Introduction .....	1081
31.1.1 Performance Counters and Architectural Clock Gating .....	1082
31.2 Performance Counters Registers .....	1082
31.3 Countable Conditions Index Register, CC_INDEX .....	1084
31.4 Countable Conditions Name0 Register, CC_NAME0 .....	1085
31.5 Countable Conditions Name1 Register, CC_NAME1 .....	1086
31.6 Count-Value Registers, PCT_COUNTL .....	1087
31.7 Count-Value Registers, PCT_COUNTH .....	1088
31.8 Snapshot-Value Registers, PCT_SNAPL .....	1089
31.9 Snapshot-Value Registers, PCT_SNAPH .....	1090
31.10 Configuration Register, PCT_CONFIG .....	1091
31.11 Control Register: PCT_CONTROL .....	1092
31.12 Index-Select Register: PCT_INDEX .....	1094
31.13 Minimum Value Registers, PCT_MINMAXL .....	1095
31.14 Maximum Value Registers, PCT_MINMAXH .....	1096
31.15 Address-Range Registers, PCT_RANGEL .....	1097
31.16 Address-Range Registers, PCT_RANGEH .....	1098

31.17 User-Flag Register, PCT_UFLAGS .....	1099
31.18 Performance Counter Build-Configuration Register, PCT_BUILD .....	1101
31.19 Countable Conditions Build Configuration Register, CC_BUILD .....	1102
<b>Chapter 32</b>	
ARC RTT .....	1103
32.1 ARC RTT Introduction .....	1103
32.2 ARC RTT Auxiliary Registers .....	1103
32.2.1 ARC RTT Address Register, RTT_ADDRESS .....	1104
32.2.2 ARC RTT DATA Register, RTT_DATA .....	1105
32.2.3 ARC RTT CMD Register, RTT_COMMAND .....	1106
32.2.4 ARC RTT Build Configuration Register, RTT_BUILD .....	1107
<b>Chapter 33</b>	
Power Domain Management and DVFS .....	1109
33.1 Power Domain Management and DVFS Introduction .....	1109
33.2 PDM and DVFS Register Interface .....	1109
33.3 Core Power Status Register, PDM_PSTAT .....	1110
33.4 ARC RTT Power Status Register, RTT_PDM_PSTAT .....	1112
33.5 Core Performance Register, DVFS_PL .....	1113
33.6 Power Down Register, PDM_PMODE .....	1114
33.7 Build Configuration Registers .....	1114
33.8 Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD .....	1115
<b>Chapter 34</b>	
Floating-Point Unit .....	1117
34.1 Core Register Set .....	1117
34.2 Extension Core Registers .....	1118
34.3 FPU Auxiliary Registers .....	1119
34.4 Floating-Point Unit Control Register, FPU_CTRL .....	1120
34.5 Floating-Point Unit Status Register, FPU_STATUS .....	1121
34.6 Double-precision Floating Point D1 Lower Register, AUX_DPFP1L .....	1124
34.7 Double-precision Floating Point D1 Higher Register, AUX_DPFP1H .....	1125
34.8 Double-precision Floating Point D2 Lower Register, AUX_DPFP2L .....	1126
34.9 Double-precision Floating Point D2 Higher Register, AUX_DPFP2H .....	1127
34.10 Build Configuration Registers .....	1127
34.11 Floating-Point Unit Build Register, FPU_BUILD .....	1128
<b>Chapter 35</b>	
XY .....	1129
35.1 XY Data Organization .....	1130
35.1.1 Address Generation .....	1130
35.1.2 Data Transformation .....	1132
35.2 XY Extension Core Registers .....	1133
35.3 XY Register Set .....	1134
35.3.1 XCCM Base Address, XCCM_BASE .....	1135
35.3.2 YCCM Base Address, YCCM_BASE .....	1136
35.3.3 XY Build Configuration Register, XY_BUILD .....	1137
<b>Chapter 36</b>	

Address Generation Unit Auxiliary Interface .....	1139
36.1 AGU Auxiliary Register Set .....	1141
36.1.1 AGU Address Pointer Registers, AGU_AUX_APx .....	1142
36.1.2 AGU Offset Registers, AGU_AUX_OSx .....	1147
36.1.3 AGU Modifier Registers, AGU_AUX_MODx .....	1148
36.1.4 AGU Build Configuration Register, AGU_BUILD .....	1154
Chapter 37 .....	
Bitstream Processing .....	1157
37.0.1 Bitstream Control Register, BS_AUX_CTRL .....	1158
37.0.2 Bitstream Base Address Register, BS_AUX_ADDR .....	1159
37.0.3 Bitstream Offset Register, BS_AUX_BIT_OS .....	1160
37.0.4 Bitstream Write Data Register, BS_AUX_WDATA .....	1161
37.1 Build Configuration Registers .....	1161
37.1.1 Bitstream Identity Register, BS_BUILD .....	1162
37.2 Exceptions .....	1162
Chapter 38 .....	
User Auxiliary Register Interface .....	1163
38.1 UAUX in Secure Mode .....	1163
Chapter 39 .....	
ARConnect .....	1165
39.1 ARConnect Introduction .....	1165
39.2 ARConnect Register Interface .....	1165
39.3 ARConnect Command Register, MCIP_CMD .....	1167
39.4 ARConnect Write Data Register, MCIP_WDATA .....	1168
39.5 ARConnect Read Data Register, MCIP_READBACK .....	1169
39.5.1 ARConnect Build Configuration Register, MCIP_SYSTEM_BUILD .....	1170
39.5.2 Inter-core Semaphore Unit Build Configuration Register, MCIP_SEMA_BUILD .....	1172
39.5.3 Inter-core Message Unit Build Configuration Register, MCIP_MESSAGE_BUILD .....	1173
39.5.4 Power Management Unit Build Configuration Register, MCIP_PMU_BUILD .....	1174
39.5.5 ARConnect Global Free Running Counter Build Configuration Register, MCIP_GFRC_BUILD .....	1175
39.5.6 Inter-Core Interrupt Unit Build Configuration Register, MCIP_ICI_BUILD .....	1176
39.5.7 Inter-Core Debug Unit Build Configuration Register, MCIP_ICD_BUILD .....	1177
39.5.8 ARConnect Power Domain Management Build Configuration Register, MCIP_PDM_BUILD .....	1178
39.6 Programming Restrictions .....	1178
39.6.1 Atomic Operation .....	1178
39.6.2 Preventing Conflicts .....	1178
Chapter 40 .....	
micro DMA Controller .....	1181
40.1 DMAC Features .....	1181
40.2 DMAC Auxiliary Register Interface .....	1181
40.2.1 XY DMA Controller .....	1182
40.2.2 DMA Controller Configuration Register, DMACTRL .....	1184
40.2.3 DMA Channel Enable Register, DMACENB .....	1185
40.2.4 DMA Channel Disable Register, DMACDSB .....	1186

40.2.5 DMA Channel High Priority Level Register, DMACHPRI .....	1187
40.2.6 DMA Channel Normal Priority Level Register, DMACNPRI .....	1188
40.2.7 DMA Channel Transfer Request Register, DMACREQ .....	1189
40.2.8 DMA Channel Status Register, DMACSTAT0 .....	1190
40.2.9 DMA Channel Status Register, DMACSTAT1 .....	1191
40.2.10 DMA Channel Interrupt Request Status Register, DMACIRQ .....	1193
40.2.11 DMA Channel Structure Base Address Register, DMACBASE .....	1194
40.2.12 DMA Channel Control Register, DMACTRLx .....	1197
40.2.13 DMA Channel Source Address Register, DMASARx .....	1203
40.2.14 DMA Channel Destination Address Register, DMADARx .....	1204
40.2.15 DMA Channel Linked-List Pointer Register, DMALLPx .....	1206
40.2.16 DMA Channel Reset Register, DMACRST .....	1209
40.2.17 DMA Build Configuration Register, DMAC_BUILD .....	1210
<b>Part 4</b>	
<b>ARCv2 Scalar DSP Extensions .....</b>	<b>1213</b>
<b>Chapter 41</b>	
Scalar DSP Introduction .....	1215
41.1 Key Features of the ARCv2 ISA DSP Extensions .....	1215
41.2 Programmer's Model .....	1216
41.3 Configurability .....	1216
41.3.1 Programming Model Options .....	1217
41.3.2 ISA Options .....	1218
<b>Chapter 42</b>	
Data Organization and Addressing .....	1219
42.1 DSP Data Formats .....	1219
42.1.1 8-Bit Signed Integer Vector .....	1219
42.1.2 16-Bit Signed Integer .....	1219
42.1.3 16-Bit Unsigned Integer .....	1219
42.1.4 16-Bit Signed Fractional .....	1220
42.1.5 16-Bit Signed Integer Vector .....	1220
42.1.6 16-Bit Unsigned Integer Vector .....	1220
42.1.7 16-Bit Signed Fractional Vector .....	1220
42.1.8 Complex 16+16 Bit Signed Fractional Data .....	1221
42.1.9 32-Bit Signed Integer .....	1221
42.1.10 32-Bit Unsigned Integer .....	1221
42.1.11 32-Bit Signed Fractional .....	1221
42.1.12 Accumulators .....	1222
42.1.13 Accumulator Saturation .....	1224
42.1.14 Accumulator Operations .....	1224
<b>Chapter 43</b>	
DSP Auxiliary Interface .....	1227
43.1 Extension Core Registers .....	1227
43.2 Auxiliary Register Set .....	1228
43.2.1 DSP Auxiliary Registers .....	1228
43.2.2 Accumulator Low Register, ACC0_LO .....	1229
43.2.3 Low Accumulator Guard and Status Register, ACC0_GLO .....	1230

43.2.4 Accumulator High Register, ACC0_HI .....	1231
43.2.5 High Accumulator Guard and Status Register, ACC0_GHI .....	1232
43.2.6 DSP Butterfly Instructions Data Register, DSP_BFLY0 .....	1233
43.2.7 DSP FFT Control Register, DSP_FFT_CTRL .....	1234
43.2.8 DSP Control Register, DSP_CTRL .....	1235
43.3 Build Configuration Registers .....	1236
43.3.1 DSP Build Configuration Register, DSP_BUILD .....	1237
 Chapter 44	
Scalar DSP Instruction Set Details .....	1239
44.1 Notations Used in the DSP Instructions .....	1239
44.2 List of Instructions .....	1239
ABSS .....	1246
ABSSH .....	1249
ADCS .....	1252
ADDS .....	1254
ASLACC .....	1257
ASLS .....	1261
ASLSACC .....	1264
ASRS .....	1269
ASRSR .....	1272
CBFLYHF0R .....	1275
CBFLYHF1R .....	1279
CMACCHFR .....	1283
CMACCHNFR .....	1287
CMACHFR .....	1290
CMACHNFR .....	1294
CMPYCHFR .....	1297
CMPYCHNFR .....	1301
CMPYHFR .....	1304
CMPYHFMR .....	1309
CMPYHNFR .....	1313
DIV .....	1316
DIVF .....	1319
DIVU .....	1322
DMACH .....	1325
DMACHBL .....	1328
DMACHBM .....	1331
DMACHF .....	1334
DMACHFR .....	1337
DMACHU .....	1340
DMPYHBL .....	1346
DMPYHBM .....	1349
DMPYHF .....	1352
DMPYHFR .....	1355
DMPYHU .....	1358
DMPYHWF .....	1361
GETACC .....	1367
MAC .....	1372

MACD .....	1375
MACDF .....	1378
MACDU .....	1381
MACF .....	1384
MACFR .....	1387
MACU .....	1390
MACWHFM .....	1393
MACWHFMR .....	1396
MACWHL .....	1399
MACWHFL .....	1402
MACWHFLR .....	1405
MACWHKL .....	1408
MACWHKUL .....	1411
MACWHUL .....	1414
MODIF .....	1417
MODAPP .....	1421
MPY .....	1423
MPY_S .....	1426
MPYD .....	1428
MPYDF .....	1431
MPYDU .....	1434
MPYF .....	1437
MPYFR .....	1440
MPYM MPYH .....	1443
MPYMU MPYHU .....	1446
MPYU .....	1449
MPYUW .....	1452
MPYUW_S .....	1455
MPYW .....	1457
MPYW_S .....	1460
MPYWHFL .....	1462
MPYWHFLR .....	1465
MPYWHFM .....	1468
MPYWHFMR .....	1471
MPYWHL .....	1474
MPYWHUL .....	1477
MPYWHKL .....	1480
MPYWHKUL .....	1483
MSUBDF .....	1486
MSUBF .....	1489
MSUBFR .....	1492
MSUBWHFL .....	1495
MSUBWHFLR .....	1498
MSUBWHFM .....	1501
MSUBWHFMR .....	1504
NEGS .....	1507
NEGSH .....	1510
NORMACC .....	1513
QMACH .....	1517

REMU .....	1525
RNDH .....	1528
SATH .....	1531
SATF .....	1534
SBCS .....	1536
SETACC .....	1539
SQRT .....	1544
SQRTF .....	1546
SUBS .....	1548
VABS2H .....	1551
VABSS2H .....	1554
VADD2H .....	1557
VADD4B .....	1560
VADDS2H .....	1563
VADDSUB2H .....	1566
VADDSUBS2H .....	1569
VALGN2H .....	1572
VASL2H .....	1574
VASLS2H .....	1577
VASR2H .....	1580
VASRS2H .....	1583
VASRSR2H .....	1586
VEXT2BHL .....	1589
VEXT2BHLF .....	1591
VEXT2BHM .....	1593
VEXT2BHMF .....	1595
VLSR2H .....	1597
VMAC2H .....	1600
VMAC2HF .....	1603
VMAC2HFR .....	1607
VMAC2HNFR .....	1611
VMAC2HU .....	1614
VMAX2H .....	1617
VMIN2H .....	1620
VMPY2H .....	1623
VMPY2HF .....	1626
VMPY2HFR .....	1630
VMPY2HU .....	1634
VMPY2HWF .....	1637
VMSUB2HF .....	1641
VMSUB2HFR .....	1645
VMSUB2HNFR .....	1649
VNEG2H .....	1652
VNEGS2H .....	1655
VNORM2H .....	1658
VPACK2HL .....	1661
VPACK2HM .....	1663
VPACK2HBL .....	1665
VPACK2HBLF .....	1667

VPACK2HBM .....	1669
VPACK2HBMF .....	1671
VPERM .....	1673
VREP2HL .....	1676
VREP2HM .....	1678
VSEXT2BHL .....	1680
VSEXT2BHM .....	1682
VSUB2H .....	1684
VSUBS2H .....	1687
VSUBS4H .....	1690
VSUB4B .....	1693
VSUBADD2H .....	1696
VSUBADDS2H .....	1699
<b>Part 5 Appendices .....</b>	<b>1703</b>
<b>Appendix A</b>	
Build and Auxiliary Register List .....	1705
A.1 Build Configuration Registers .....	1712
<b>Appendix B</b>	
ISA Version Differences .....	1715
<b>Appendix C</b>	
Implementation-dependent Behavior .....	1723
C.1 EM Exception Handling .....	1723
C.1.1 ECR User-mode Setting in the ARC EM Family of Cores .....	1723
C.2 Cause Codes for Memory Accesses and Data/Address Errors .....	1723
C.3 Exception Handling .....	1724
C.3.1 ECR User-mode Setting for the ARC EM Family of Cores .....	1724
C.4 Result on Overflow from Integer Division .....	1724
<b>Appendix D</b>	
Processor Component Options .....	1725
D.1 COMPONENT: com.arc.hardware.Actionpoints.1_0 .....	1725
D.2 COMPONENT: com.arc.hardware.AGU.1_0 .....	1725
D.3 COMPONENT: com.arc.hardware.ARcv2EM.1_0 .....	1726
D.4 COMPONENT: com.arc.hardware.Calibration_NVM.1_0 .....	1732
D.5 COMPONENT: com.arc.hardware.CPU_isle.1_0 .....	1732
D.6 COMPONENT: com.arc.hardware.Data_Cache.1_0 .....	1734
D.7 COMPONENT: com.arc.hardware.DCCM.1_0 .....	1735
D.8 COMPONENT: com.arc.hardware.Debug_Interface.1_0 .....	1736
D.9 COMPONENT: com.arc.hardware.DMA_Controller.1_0 .....	1736
D.10 COMPONENT: com.arc.hardware.DSP.1_0 .....	1737
D.11 COMPONENT: com.arc.hardware.Floating_point_unit.1_0 .....	1738
D.12 COMPONENT: com.arc.hardware.ICCM0.1_0 .....	1739
D.13 COMPONENT: com.arc.hardware.ICCM1.1_0 .....	1740
D.14 COMPONENT: com.arc.hardware.Instruction_Cache.1_0 .....	1740
D.15 COMPONENT: com.arc.hardware.Instruction_Fetch_Queue.1_0 .....	1742

D.16 COMPONENT: com.arc.hardware.Interrupt_Controller.1_0 .....	1742
D.17 COMPONENT: com.arc.hardware.Key_NVM.1_0 .....	1743
D.18 COMPONENT: com.arc.hardware.Lockstep_Comparator.1_0 .....	1743
D.19 COMPONENT: com.arc.hardware.Memory_Protection_Unit.1_0 .....	1744
D.20 COMPONENT: com.arc.hardware.Performance_Monitor.1_0 .....	1744
D.21 COMPONENT: com.arc.hardware.Program_NVM.1_0 .....	1744
D.22 COMPONENT: com.arc.hardware.Real_time_trace_producer.1_0 .....	1745
D.23 COMPONENT: com.arc.hardware.Secure_pipeline_features.1_0 .....	1745
D.24 COMPONENT: com.arc.hardware.Secure_Timer_0.1_0 .....	1746
D.25 COMPONENT: com.arc.hardware.Secure_Timer_1.1_0 .....	1746
D.26 COMPONENT: com.arc.hardware.SmaRT.1_0 .....	1746
D.27 COMPONENT: com.arc.hardware.System.1_0 .....	1747
D.28 COMPONENT: com.arc.hardware.Timer_0.1_0 .....	1748
D.29 COMPONENT: com.arc.hardware.Timer_1.1_0 .....	1748
D.30 COMPONENT: com.arc.hardware.Watchdog_Timer.1_0 .....	1748
D.31 COMPONENT: com.arc.hardware.XY_DMA_Controller.1_0 .....	1750
D.32 COMPONENT: com.arc.hardware.ARConnect.1_0 .....	1750
D.33 COMPONENT: com.arc.hardware.AR Cv2MSS.BusFabric.1_0 .....	1752
D.34 COMPONENT: com.arc.hardware.AR Cv2MSS.ClkCtrl.1_0 .....	1753
D.35 COMPONENT: com.arc.hardware.AR Cv2MSS.DummyMST.1_0 .....	1753
D.36 COMPONENT: com.arc.hardware.AR Cv2MSS.DummySLV.1_0 .....	1755
D.37 COMPONENT: com.arc.hardware.AR Cv2MSS.Profiler.1_0 .....	1758
D.38 COMPONENT: com.arc.hardware.AR Cv2MSS.SnoopTrafficGen.1_0 .....	1759
D.39 COMPONENT: com.arc.hardware.AR Cv2MSS.SRAMCtrl.1_0 .....	1759
D.40 COMPONENT: com.arc.hardware.implementation.1_0 .....	1762
Appendix E	
Auxiliary Functions for DSP and XY Extensions .....	1767
E.1 Auxiliary Function that implements the saturation and rounding functionality .....	1767
E.2 Data Transformation .....	1768
E.2.1 Realigns Data from Even and Odd XY Banks .....	1768
E.2.2 Data Mask: Masks Unused MSB Bytes .....	1768
E.2.3 Data to Container Expansion Function .....	1768
E.2.4 ReplicateReverse Function .....	1769
E.2.5 Destination Transformation .....	1770
E.2.6 Destination Memory Function .....	1771
E.3 Container to Data Packing .....	1772
E.3.1 Destination Transformation for Endian Parameters .....	1773
E.3.2 Destination Transformation for Endian Parameters .....	1774

# List of Figures

Figure 2-1 Address Space Model .....	79
Figure 2-2 64-bit Element As a 64-bit Operand .....	80
Figure 2-3 32-bit Elements in a 64-bit Operand .....	80
Figure 2-4 16-bit Elements in a 64-bit Operand .....	80
Figure 2-5 Register Pair Holding Operands in Little-endian Mode .....	80
Figure 2-6 Register Pair Holding Operands in Big-endian Mode .....	81
Figure 2-7 32-Bit Elements in a 32-bit Operand .....	82
Figure 2-8 16-Bit Elements in a 32-bit Operand .....	82
Figure 2-9 Expansion of a u6 Literal to a 32-Bit Word (w-shimm) .....	82
Figure 2-10 Expansion of an s12 Literal to a 32-Bit Word (w-shimm) .....	82
Figure 2-11 Expansion of a u6 Literal to Half-Word (16-bit, hw-shimm) .....	82
Figure 2-12 Expansion of an s12 Literal to Half-Word (16-bit, hw-shimm) .....	82
Figure 2-13 A Half-Word (16-bit) Short Limm Value Replicated to form a 64-Bit Operand .....	82
Figure 2-14 A word (32-bit) Short-limm or Limm value Replicated to form a 64-Bit Operand .....	83
Figure 2-15 Two 16-Bit Vectors with Expanded u6 Operands .....	83
Figure 2-16 Two 16-Bit Vectors with Expanded s12 Operands .....	83
Figure 2-17 Four 8-Bit Vectors with Expanded u6 Operands .....	83
Figure 2-18 Four 16-Bit Vectors with Expanded s12 Operands .....	83
Figure 2-19 Four 8-Bit Vectors with s12 Operands .....	83
Figure 2-20 Expansion of a w6 in STD .....	83
Figure 2-21 Expansion of a limm in STD .....	83
Figure 2-22 Single-Precision Floating-Point Data .....	84
Figure 2-23 Double-Precision Floating-Point Data .....	84

Figure 2-24 Double Precision Floating Point Data.....	85
Figure 2-25 64-bit Register Data in Byte-Wide Memory, Little-Endian .....	88
Figure 2-26 64-bit Register Data in Byte-Wide Memory, Big-Endian .....	88
Figure 2-27 Register containing 32-bit Data .....	88
Figure 2-28 32-bit Register Data in Byte-Wide Memory, Little-Endian .....	88
Figure 2-29 32-bit Register Data in Byte-Wide Memory, Big-Endian .....	89
Figure 2-30 Register Containing 16-Bit Data.....	89
Figure 2-31 16-bit Register Data in Byte-Wide Memory .....	89
Figure 2-32 16-bit Register Data in Byte-Wide Memory, Big-Endian .....	89
Figure 2-33 Register Containing 8-Bit Data.....	90
Figure 2-34 8-bit Register Data in Byte-Wide Memory.....	90
Figure 2-35 Register Containing 1-Bit Data.....	90
Figure 2-36 Half-word Numbering of a 32-Bit Instruction or Long-immediate Data Item .....	90
Figure 2-37 Little-endian Offset of each Byte in a 32-Bit Instruction or Immediate .....	91
Figure 2-38 Big-endian Offset of each Byte in a 32-Bit Instruction or Immediate.....	91
Figure 2-39 Little-endian Memory Offset of each Byte in a 16-Bit Instruction .....	91
Figure 2-40 Big-endian Memory Offset of each Byte in a 16-bit Instruction .....	91
Figure 3-1 Core Register Map Summary.....	95
Figure 3-2 PCL Register .....	101
Figure 3-3 ACCH for Double-Precision FPU operations .....	102
Figure 3-4 ACCL for Double-Precision FPU operations.....	102
Figure 3-5 ACCL for Single-Precision FPU Operations .....	102
Figure 3-6 LP_START Register (PC_SIZE == 32).....	109
Figure 3-7 LP_START Register (PC_SIZE < 32) .....	109
Figure 3-8 LP_END Register .....	110
Figure 3-9 LP_END Register (PC_SIZE < 32) .....	110
Figure 3-10 IDENTITY Register .....	111
Figure 3-11 PC Register Addressing Full 32-bit Address Space .....	113
Figure 3-12 PC Register Addressing a Reduced Address Space (PC_SIZE == 32).....	113

Figure 3-13 SEC_STAT .....	114
Figure 3-14 STATUS32 Register .....	117
Figure 3-15 STATUS32_P0 Register .....	124
Figure 3-16 AUX_USER_SP Register .....	125
Figure 3-17 AUX_IRQ_CTRL Register .....	126
Figure 3-18 INT_VECTOR_BASE Register (PC_SIZE == 32) .....	128
Figure 3-19 INT_VECTOR_BASE Register (PC_SIZE < 32) .....	128
Figure 3-20 INT_VECTOR_BASE_S Register (PC_SIZE == 32) .....	129
Figure 3-21 INT_VECTOR_BASE_S Register (PC_SIZE < 32) .....	129
Figure 3-22 AUX_KERNEL_SP Register .....	130
Figure 3-23 AUX_SEC_U_SP Register .....	131
Figure 3-24 AUX_SEC_K_SP Register .....	132
Figure 3-25 AUX_NSEC_SP Register .....	133
Figure 3-26 AUX_IRQ_ACT Register .....	134
Figure 3-27 IRQ_PRIORITY_PENDING Register .....	135
Figure 3-28 AUX_IRQ_HINT Register .....	136
Figure 3-29 IRQ_PRIORITY Register .....	137
Figure 3-30 NSC_TABLE_TOP Register when PC_SIZE == 32 .....	139
Figure 3-31 NSC_TABLE_BASE Register when PC_SIZE == 32 .....	140
Figure 3-32 JLI_BASE Register when PC_SIZE == 32 .....	141
Figure 3-33 JLI_BASE Register when PC_SIZE < 32 .....	141
Figure 3-34 LDI_BASE Register when ADDR_SIZE == 32 .....	142
Figure 3-35 LDI_BASE Register when ADDR_SIZE < 32 .....	142
Figure 3-36 EI_BASE Register when PC_SIZE == 32 .....	143
Figure 3-37 EI_BASE Register when PC_SIZE < 32 .....	143
Figure 3-38 ERET Register when PC_SIZE == 32 .....	144
Figure 3-39 ERET Register when PC_SIZE < 32 .....	144
Figure 3-40 ERBTA Register when PC_SIZE == 32 .....	145
Figure 3-41 ERBTA Register when PC_SIZE < 32 .....	145

Figure 3-42 ERSTATUS Register . . . . .	146
Figure 3-43 ECR Register . . . . .	147
Figure 3-44 EFA Register . . . . .	148
Figure 3-45 ERSEC_STAT . . . . .	150
Figure 3-46 AUX_SEC_EXCEPT . . . . .	152
Figure 3-47 ICAUSE Register . . . . .	154
Figure 3-48 IRQ_SELECT Register . . . . .	155
Figure 3-49 IRQ_ENABLE Register . . . . .	156
Figure 3-50 IRQ_TRIGGER Register . . . . .	157
Figure 3-51 IRQ_STATUS Register . . . . .	158
Figure 3-52 XPU Register . . . . .	159
Figure 3-53 EFA_EXT Register . . . . .	161
Figure 3-54 BTA Register (PC_SIZE == 32) . . . . .	162
Figure 3-55 BTA Register (PC_SIZE < 32) . . . . .	162
Figure 3-56 IRQ_PULSE_CANCEL Register . . . . .	164
Figure 3-57 IRQ_PENDING Register . . . . .	165
Figure 3-58 User Extension Flags Register, XFLAGS . . . . .	166
Figure 3-59 BCR_VER Register . . . . .	168
Figure 3-60 BTA_LINK_BUILD Register . . . . .	169
Figure 3-61 VECBASE_AC_BUILD Register . . . . .	170
Figure 3-62 SEC_VECBASE_BUILD Register . . . . .	171
Figure 3-63 RF_BUILD Register . . . . .	172
Figure 3-64 MULTIPLY_BUILD Register . . . . .	174
Figure 3-65 SWAP_BUILD Register . . . . .	175
Figure 3-66 NORM_BUILD Register . . . . .	176
Figure 3-67 MINMAX_BUILD Register . . . . .	177
Figure 3-68 BARREL_BUILD Register . . . . .	178
Figure 3-69 ISA_CONFIG Build Register . . . . .	179
Figure 3-70 ARCH_CONFIG Register . . . . .	181

Figure 3-71 IRQ_BUILD Register . . . . .	182
Figure 5-1 Long Immediate source data value . . . . .	260
Figure 6-1 Branch Conditionally Format . . . . .	310
Figure 6-2 Branch Unconditional Far Format . . . . .	310
Figure 6-3 Branch on Compare Register-register Format . . . . .	311
Figure 6-4 Branch on Compare Register-Immediate Format . . . . .	312
Figure 6-5 Branch and Link Conditionally Format . . . . .	313
Figure 6-6 Branch and Link Unconditional Far Format . . . . .	314
Figure 6-7 Load Register with Offset Format . . . . .	315
Figure 6-8 Store Register with Offset Format . . . . .	316
Figure 6-9 Store Register with Offset Format . . . . .	316
Figure 6-10 General Operations Register-Register Format . . . . .	318
Figure 6-11 General Operations Register with Unsigned 16-bit Immediate Format . . . . .	319
Figure 6-12 General Operations Register with Signed 12-bit Immediate Format . . . . .	319
Figure 6-13 General Operations Conditional Register Format . . . . .	319
Figure 6-14 General Operations Conditional Register with Unsigned 6-bit Immediate . . . . .	320
Figure 6-15 Load Register-Register Format . . . . .	323
Figure 6-16 Extension ALU Operation, register-register . . . . .	330
Figure 6-17 Extension ALU Operation, register with unsigned 6-bit immediate . . . . .	330
Figure 6-18 Extension ALU Operation, register with signed 12-bit immediate . . . . .	330
Figure 6-19 Extension ALU Operation, conditional register . . . . .	330
Figure 6-20 Extension ALU Operation, cc register with unsigned 6-bit immediate . . . . .	331
Figure 7-1 Compact Move Format . . . . .	337
Figure 7-2 Compact Load Format . . . . .	337
Figure 7-3 Compact Load and Subtract Format . . . . .	338
Figure 7-4 Compact Add Format . . . . .	338
Figure 7-5 Compact Load and Store Format . . . . .	339
Figure 7-6 Compact Load Indexed Format . . . . .	339
Figure 7-7 Indexed Jump and Execute Format . . . . .	340

Figure 7-8 Load and Add Register-Register Format . . . . .	341
Figure 7-9 Add/Sub/Shift Register-Immediate Format . . . . .	342
Figure 7-10 Dual Register with a 3-bit b Register Format . . . . .	343
Figure 7-11 Dual Register with a 3-bit Biased Signed Integer Format . . . . .	343
Figure 7-12 Long Immediate Operand Format . . . . .	344
Figure 7-13 Compact General Operations Register-Register Format . . . . .	345
Figure 7-14 Compact Single Operand, Jumps, Special Formats . . . . .	346
Figure 7-15 Compact Load/Store with Offset Format . . . . .	349
Figure 7-16 Compact Shift/Sub Bit Immediate Format . . . . .	350
Figure 7-17 Compact Load/Add GP-Relative Format . . . . .	352
Figure 7-18 Compact Load PCL Relative Format . . . . .	353
Figure 7-19 Compact Move Immediate Format . . . . .	354
Figure 7-20 Compact ADD/CMP Immediate Format . . . . .	355
Figure 7-21 Compact Branch on Compare Register with Zero Format . . . . .	356
Figure 7-22 Compact Branch Conditionally Format . . . . .	357
Figure 7-23 Branch Conditionally with cc Field Format . . . . .	357
Figure 7-24 Compact Branch and Link Unconditionally Format . . . . .	359
Figure 8-1 Loop Detection and Update Mechanism . . . . .	643
Figure 8-2 SLEEP Operand . . . . .	777
Figure 9-1 Example Host Memory Maps, Contiguous Host Memory . . . . .	849
Figure 9-2 Example Host Memory Maps, Host Memory and Host I/O . . . . .	849
Figure 11-1 SecureShield Overview . . . . .	857
Figure 12-1 SEC_BUILD Register . . . . .	864
Figure 13-1 MPU_INDEX for 16 Regions . . . . .	873
Figure 13-2 MPU_INDEX for 2 Regions . . . . .	873
Figure 13-3 MPU_RSTART . . . . .	874
Figure 13-4 MPU_RENDER . . . . .	875
Figure 13-5 MPU_RPER . . . . .	876
Figure 13-6 MPU_PROBE . . . . .	879

Figure 14-1 AUX_KEY_DATA Register .....	885
Figure 14-2 AUX_KEY_UNLOCK Register .....	886
Figure 15-1 ERP_CTRL Register .....	888
Figure 15-2 RFERP_STATUS_0 Register .....	891
Figure 15-3 RFERP_STATUS_1 Register .....	893
Figure 15-4 ESYN Register .....	894
Figure 15-5 ERP_BUILD Register .....	895
Figure 16-1 Example 1: MPU_RDB0 Register .....	904
Figure 16-2 Example 1: MPU_RDP0 Register .....	904
Figure 16-3 Example 1: MPU_RDB1 Register .....	904
Figure 16-4 Example 1: MPU_RDP1 Register .....	904
Figure 16-5 Example 1: MPU_RDB2 Register .....	904
Figure 16-6 Example 1: MPU_RDP2 Register .....	904
Figure 16-7 Overlapping Regions Memory Map .....	905
Figure 16-8 Example 2: MPU_RDB0 Register .....	905
Figure 16-9 Example 2: MPU_RDP0 Register .....	905
Figure 16-10 Example 2: MPU_RDB7 Register .....	905
Figure 16-11 Example 2: MPU_RDP7 Register .....	905
Figure 16-12 Example 1: MPU_RDB0 Register .....	906
Figure 16-13 Example 1: MPU_RDP0 Register .....	906
Figure 16-14 Example 1: MPU_RDB1 Register .....	906
Figure 16-15 Example 1: MPU_RDP1 Register .....	906
Figure 16-16 Example 1: MPU_RDB2 Register .....	906
Figure 16-17 Example 1: MPU_RDP2 Register .....	906
Figure 16-18 Overlapping Regions Memory Map .....	907
Figure 16-19 Example 2: MPU_RDB0 Register .....	907
Figure 16-20 Example 2: MPU_RDP0 Register .....	907
Figure 16-21 Example 2: MPU_RDB7 Register .....	907
Figure 16-22 Example 2: MPU_RDP7 Register .....	907

Figure 16-23 MPU_EN Register .....	913
Figure 16-24 MPU_ECR Register .....	916
Figure 16-25 MPU Region Descriptor Base Registers .....	918
Figure 16-26 MPU Region Descriptor Permissions Registers.....	919
Figure 16-27 MPU_BUILD Register .....	921
Figure 17-1 Stack regions.....	926
Figure 17-2 Stack Checking Scenarios .....	927
Figure 17-3 USTACK_TOP when ADDR_SIZE == 32.....	929
Figure 17-4 USTACK_TOP when ADDR_SIZE<32.....	929
Figure 17-5 KSTACK_TOP when ADDR_SIZE == 32.....	929
Figure 17-6 KSTACK_TOP, when ADDR_SIZE < 32 .....	929
Figure 17-7 USTACK_BASE when ADDR_SIZE == 32.....	930
Figure 17-8 USTACK_BASE when ADDR_SIZE < 32.....	930
Figure 17-9 KSTACK_BASE when ADDR_SIZE == 32.....	930
Figure 17-10 KSTACK_BASE when ADDR_SIZE < 32.....	930
Figure 17-11 S_USTACK_TOP when ADDR_SIZE == 32.....	932
Figure 17-12 S_USTACK_TOP when ADDR_SIZE<32.....	932
Figure 17-13 S_KSTACK_TOP when ADDR_SIZE == 32.....	932
Figure 17-14 S_KSTACK_TOP, when ADDR_SIZE < 32 .....	932
Figure 17-15 S_USTACK_BASE when ADDR_SIZE == 32.....	933
Figure 17-16 S_USTACK_BASE when ADDR_SIZE < 32.....	933
Figure 17-17 S_KSTACK_BASE when ADDR_SIZE == 32.....	933
Figure 17-18 S_KSTACK_BASE when ADDR_SIZE < 32.....	933
Figure 17-19 STACK_REGION_BUILD Register .....	934
Figure 17-20 CPROT_BUILD Register.....	938
Figure 18-1 AUX_ICCM, base address for ICCM (ADDR_SIZE == 32) .....	940
Figure 18-2 AUX_ICCM, base address for ICCM (ADDR_SIZE == 16) .....	940
Figure 18-3 AUX_ICCM, base address for ICCM (ADDR_SIZE == m) .....	940
Figure 18-4 ICCM_BUILD Register .....	941

Figure 19-1 IFQUEUE_BUILD Register.....	943
Figure 20-1 IC_IVIC Register .....	947
Figure 20-2 IC_CTRL Register .....	948
Figure 20-3 IC_LIL Register.....	949
Figure 20-4 IC_IVIL Register.....	951
Figure 20-5 IC_TAG Register for Reads .....	953
Figure 20-6 IC_TAG Register for Writes .....	953
Figure 20-7 IC_XTAG Register .....	955
Figure 20-8 IC_DATA Register.....	956
Figure 20-9 I_CACHE_BUILD Register.....	957
Figure 21-1 AUX_DCCM, base address for DCCM (ADDR_SIZE == 32) .....	960
Figure 21-2 AUX_DCCM, base address for DCCM (ADDR_SIZE == 16) .....	960
Figure 21-3 AUX_DCCM, base address for DCCM (ADDR_SIZE == m) .....	960
Figure 21-4 DCCM_BUILD Register .....	961
Figure 22-1 DC_IVDC Register.....	965
Figure 22-2 DC_CTRL Register.....	967
Figure 22-3 DC_LDL Register.....	968
Figure 22-4 DC_IVDL Register .....	971
Figure 22-5 DC_FLSH Register.....	973
Figure 22-6 DC_FLDL Register.....	975
Figure 22-7 DC_RAM_ADDR.....	977
Figure 22-8 DC_TAG for Reads .....	978
Figure 22-9 DC_TAG for Writes .....	978
Figure 22-10 DC_XTAG Register .....	980
Figure 22-11 DC_DATA.....	981
Figure 22-12 AUX_CACHE_LIMIT, start of non-cached regions (ADDR_SIZE == 32).....	982
Figure 22-13 AUX_CACHE_LIMIT, (ADDR_SIZE == 16) .....	982
Figure 22-14 AUX_CACHE_LIMIT, start of non-cached regions (ADDR_SIZE == m) .....	982
Figure 22-15 Volatile Space .....	983

Figure 22-16 D_CACHE_BUILD .....	984
Figure 23-1 NV_ICCM_CTRL Register .....	988
Figure 23-2 NV_ICCM_ERASE Register .....	990
Figure 23-3 NV_ICCM_PROG Register.....	991
Figure 23-4 NV_ICCM_DATA Register .....	992
Figure 24-1 KEY_STORE_CTRL Register .....	995
Figure 24-2 KEY_STORE_PWD Register.....	997
Figure 24-3 KEY_STORE_ADDR Register .....	998
Figure 24-4 KEY_STORE_DATA Register .....	999
Figure 24-5 KEY_STORE_SID Register .....	1000
Figure 24-6 KEY_STORE_BUILD Register .....	1001
Figure 25-1 CAL_STORE_CTRL Register .....	1004
Figure 25-2 CAL_STORE_ADDR Register .....	1006
Figure 25-3 CAL_STORE_DATA Register .....	1007
Figure 25-4 CAL_STORE_BUILD Register .....	1008
Figure 26-1 Interrupt Generated after Timer Reaches Limit Value.....	1010
Figure 26-2 Timer 0 Count Value Register .....	1012
Figure 26-3 Timer 0 Control Register.....	1013
Figure 26-4 Timer 0 Limit Value Register .....	1014
Figure 26-5 Timer 1 Count Value Register .....	1015
Figure 26-6 Timer 1 Control Register.....	1016
Figure 26-7 Timer 1 Limit Value Register .....	1017
Figure 26-8 Secure Timer 0 Count Value Register .....	1018
Figure 26-9 Secure Timer 0 Control Register .....	1019
Figure 26-10 Timer 0 Control Register.....	1019
Figure 26-11 Timer 0 Limit Value Register .....	1020
Figure 26-12 Timer 1 Count Value Register .....	1021
Figure 26-13 Timer 1 Control Register.....	1022
Figure 26-14 Timer 1 Control Register.....	1022

Figure 26-15 Timer 1 Limit Value Register . . . . .	1023
Figure 26-16 AUX_RTC_CTRL Register . . . . .	1024
Figure 26-17 State Machine of A1 and A0 bits in RTC Control Register . . . . .	1025
Figure 26-18 RTC Count Low Register . . . . .	1026
Figure 26-19 AUX_RTC_HIGH Register . . . . .	1027
Figure 26-20 TIMER_BUILD Register . . . . .	1028
Figure 26-21 TIMER_BUILD Register . . . . .	1028
Figure 26-22 WDT_PASSWD Register . . . . .	1030
Figure 26-23 WDT_CTRL Register . . . . .	1031
Figure 26-24 WDT_PERIOD Register . . . . .	1032
Figure 26-25 WDT_COUNT Register . . . . .	1033
Figure 27-1 LSC_CTRL Register . . . . .	1038
Figure 27-2 LSC_MCD Register . . . . .	1040
Figure 27-3 LSC_STATUS . . . . .	1041
Figure 27-4 LSC_ERROR_REG . . . . .	1042
Figure 27-5 LSC_BUILD_AUX . . . . .	1044
Figure 29-1 DEBUG Register . . . . .	1048
Figure 29-2 DEBUGI Register . . . . .	1052
Figure 29-3 ACG_CTRL Register . . . . .	1054
Figure 30-1 Actionpoint Build Register . . . . .	1060
Figure 30-2 Actionpoint 0 Match Value Register . . . . .	1061
Figure 30-3 Actionpoint 0 Match Mask Register . . . . .	1061
Figure 30-4 Actionpoint 0 Control Register . . . . .	1062
Figure 30-5 Actionpoint 1 Match Value Register . . . . .	1066
Figure 30-6 Actionpoint 1 Match Mask Register . . . . .	1067
Figure 30-7 Actionpoint 1 Control Register . . . . .	1067
Figure 30-8 Actionpoint 2 Match Value Register . . . . .	1067
Figure 30-9 Actionpoint 2 Match Mask Register . . . . .	1067
Figure 30-10 Actionpoint 3 Match Value Register . . . . .	1068

Figure 30-11 Actionpoint 3 Match Mask Register .....	1068
Figure 30-12 Actionpoint 3 Control Register .....	1069
Figure 30-13 Actionpoint 4 Match Value Register .....	1069
Figure 30-14 Actionpoint 4 Match Mask Register .....	1069
Figure 30-15 Actionpoint 4 Control Register .....	1069
Figure 30-16 Actionpoint 5 Match Value Register .....	1070
Figure 30-17 Actionpoint 5 Match Mask Register .....	1071
Figure 30-18 Actionpoint 5 Control Register .....	1071
Figure 30-19 Actionpoint 6 Match Value Register .....	1071
Figure 30-20 Actionpoint 6 Match Mask Register .....	1072
Figure 30-21 Actionpoint 6 Control Register .....	1072
Figure 30-22 Actionpoint 7 Match Value Register .....	1073
Figure 30-23 Actionpoint 7 Match Mask Register .....	1073
Figure 30-24 Actionpoint 7 Control Register .....	1073
Figure 30-25 AP_WP_PC, Watchpoint Program Counter Register when PC_SIZE = 32 .....	1074
Figure 30-26 AP_WP_PC Register when PC_SIZE < 32 .....	1074
Figure 30-27 SmaRT Operation .....	1075
Figure 30-28 SMART_BUILD Value .....	1076
Figure 30-29 SMART_CONTROL Register .....	1077
Figure 30-30 SMART_DATA Register .....	1078
Figure 30-31 SRC_ADDR Value .....	1078
Figure 30-32 DEST_ADDR Value Register .....	1079
Figure 30-33 FLAGS Value Register .....	1079
Figure 31-1 CC_INDEX Register .....	1084
Figure 31-2 CC_NAME0 Register .....	1085
Figure 31-3 CC_NAME1 Register .....	1086
Figure 31-4 PCT_COUNTL Register .....	1087
Figure 31-5 PCT_COUNTH Register .....	1088
Figure 31-6 PCT_SNAPL Register .....	1089

Figure 31-7 PCT_SNAPH Register . . . . .	1090
Figure 31-8 PCT_CONFIG Register . . . . .	1091
Figure 31-9 PCT_CONTROL Register . . . . .	1092
Figure 31-10 PCT_INDEX Register . . . . .	1094
Figure 31-11 PCT_MINMAXL Register . . . . .	1095
Figure 31-12 PCT_MINMAXH Register . . . . .	1096
Figure 31-13 PCT_RANGEL Register . . . . .	1097
Figure 31-14 PCT_RANGEH Register . . . . .	1098
Figure 31-15 PCT_UFLAGS Register . . . . .	1099
Figure 31-16 PCT_BUILD Register . . . . .	1101
Figure 31-17 CC_BUILD Register . . . . .	1102
Figure 32-1 RTT_ADDRESS Register . . . . .	1104
Figure 32-2 RTT_DATA Register . . . . .	1105
Figure 32-3 RTT_COMMAND Register . . . . .	1106
Figure 32-4 RTT_BUILD Register . . . . .	1107
Figure 33-1 PDM_PSTAT Register . . . . .	1110
Figure 33-2 RTT_PDM_PSTAT Register . . . . .	1112
Figure 33-3 DVFS_PL Register . . . . .	1113
Figure 33-4 PDM_PMODE Register . . . . .	1114
Figure 33-5 PDM_DVFS_BUILD Register . . . . .	1115
Figure 34-1 Core Register Map Summary . . . . .	1117
Figure 34-2 ACCL for fusedMultiplyAdd and fusedMultiplySubtract Operations . . . . .	1118
Figure 34-3 FPU_CTRL Register . . . . .	1120
Figure 34-4 FPU_STATUS Register . . . . .	1121
Figure 34-5 AUX_DPFP1L, Double-precision Floating Point D1 Register, Lower Half . . . . .	1124
Figure 34-6 AUX_DPFP1H, Double-precision Floating Point D1 Register, Higher Half . . . . .	1125
Figure 34-7 AUX_DPFP2L, Double-precision Floating Point D2 Register, Lower Half . . . . .	1126
Figure 34-8 AUX_DPFP2H, Double-precision Floating Point D2 Register, Higher Half . . . . .	1127
Figure 34-9 FPU_BUILD Register . . . . .	1128

Figure 35-1 AGU Register Data Organization .....	1131
Figure 35-2 Extended Core Registers when HAS_AGU==true .....	1133
Figure 35-3 XCCM_BASE Register .....	1135
Figure 35-4 YCCM_BASE Register .....	1136
Figure 35-5 XY_BUILD Register .....	1137
Figure 36-1 Figure 5 AGU Pipeline .....	1140
Figure 36-2 AGU_AUX_APx Register .....	1142
Figure 36-3 AGU_AUX_OSx Register .....	1147
Figure 36-4 AGU_AUX_MODx Register with Offset Register Increment .....	1148
Figure 36-5 AGU_AUX_MODx Register with Immediate Offset .....	1148
Figure 36-6 AGU_AUX_MODx Register with Bit Reverse Increment using Offset Register .....	1148
Figure 36-7 AGU_AUX_MODx Register with Bit Reverse Increment using Immediate Offset .....	1148
Figure 36-8 AGU_AUX_MODx Register with Modulo Wrapping using Offset and Wrap Registers .....	1148
Figure 36-9 AGU_AUX_MODx Register with Modulo Wrapping using Immediate Offset and Wrap ..	1149
Figure 36-10 AGU_AUX_MODx Register with Modulo Wrapping using Offset Register and Immediate Wrap ..	1149
Figure 36-11 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset Register .....	1149
Figure 36-12 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset Immediate .....	1149
Figure 36-13 AGU_BUILD Register .....	1154
Figure 37-1 BS_AUX_CTRL Register .....	1158
Figure 37-2 BS_AUX_ADDR Register .....	1159
Figure 37-3 BS_AUX_BIT_OS Register .....	1160
Figure 37-4 BS_AUX_WDATA Register .....	1161
Figure 37-5 BS_BUILD Register .....	1162
Figure 39-1 MCIP_CMD Register .....	1167
Figure 39-2 MCIP_WDATA Register .....	1168
Figure 39-3 MCIP_READBACK Register .....	1169
Figure 39-4 MCIP_SYSTEM_BUILD Register .....	1170
Figure 39-5 MCIP_SEMA_BUILD Register .....	1172

Figure 39-6 MCIP_MESSAGE_BUILD Register .....	1173
Figure 39-7 CONNECT_PMU_BUILD Register .....	1174
Figure 39-8 MCIP_GFRC_BUILD Register .....	1175
Figure 39-9 MCIPICI_BUILD Register .....	1176
Figure 39-10 MCIP_ICD_BUILD Register .....	1177
Figure 39-11 MCIP_PDM_BUILD Register .....	1178
Figure 40-1 DMACTRL Register .....	1184
Figure 40-2 DMACENB Register .....	1185
Figure 40-3 DMACDSB Register .....	1186
Figure 40-4 DMACHPRI Register .....	1187
Figure 40-5 DMACNPRI Register .....	1188
Figure 40-6 DMACREQ Register .....	1189
Figure 40-7 DMASTAT0 Register .....	1190
Figure 40-8 DMASTAT1 Register .....	1191
Figure 40-9 DMAIRQ Register .....	1193
Figure 40-10 DMACBASE Register for Two DMA Channels .....	1194
Figure 40-11 DMACBASE Register for Three or Four DMA Channels .....	1194
Figure 40-12 DMACBASE Register for Five to Eight DMA Channels .....	1194
Figure 40-13 DMACBASE Register for 16 DMA Channels .....	1195
Figure 40-14 DMACTRLx Register .....	1197
Figure 40-15 DMASARx Register .....	1203
Figure 40-16 DMADARx Register .....	1204
Figure 40-17 DMALLPx Register .....	1206
Figure 40-18 DMACRST Register .....	1209
Figure 40-19 DMAC_BUILD Register .....	1210
Figure 42-1 Vector Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==0 .....	1223
Figure 42-2 Vector Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==1 .....	1223
Figure 42-3 Vector Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==0 .....	1223
Figure 42-4 Vector Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==1 .....	1223

---

Figure 42-5 Wide Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==0.....	1223
Figure 42-6 Wide Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==1.....	1224
Figure 42-7 Wide Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==0.....	1224
Figure 42-8 Wide Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==1.....	1224
Figure 43-1 ACCL for fusedMultiplyAdd and fusedMultiplySubtract Operations.....	1227
Figure 43-2 Auxiliary Register Map.....	1228
Figure 43-3 ACC0_LO Register.....	1229
Figure 43-4 ACC0_GLO Register .....	1230
Figure 43-5 ACC0_HI Register .....	1231
Figure 43-6 ACC0_GHI Register.....	1232
Figure 43-7 DSP_BFLY0 Register in Little-Endian Mode .....	1233
Figure 43-8 DSP_BFLY0 Register in Big-Endian Mode.....	1233
Figure 43-9 DSP_FFT_CTRL Register .....	1234
Figure 43-10 DSP_CTRL Register.....	1235
Figure 43-11 DSP_BUILD Register.....	1237

# List of Examples

Example 2-1 Null Instruction Format .....	93
Example 3-1 Correct LP_COUNT Setup through a Register .....	101
Example 4-1 Setting Interrupt Priority or Enable (STATUS32.IE) of an Interrupt .....	196
Example 4-2 Fast Interrupt Entry .....	200
Example 4-3 Fast Interrupt Exit .....	201
Example 4-4 Regular Interrupt Entry .....	205
Example 4-5 Regular Interrupt Exit .....	213
Example 4-6 Determine Active Interrupts .....	219
Example 4-7 Pass Control to an Exception Handler or Perform a Soft Reset .....	222
Example 4-8 Exception Vector Code .....	223
Example 4-9 Exception in a Delay Slot .....	248
Example 8-1 To obtain a semaphore using EX .....	518
Example 8-2 To Release Semaphore using ST .....	519
Example 8-3 Example Loop Code .....	642
Example 8-4 Use of Zero Overhead Loop .....	645
Example 8-5 Sleep placement in code .....	780
Example 8-6 Enable Interrupts and Sleep .....	780
Example 8-7 WEVT placement in code .....	840
Example 8-8 WLFC Placement in Code .....	843
Example 27-1 Sample Lockstep Initialization Sequence .....	1036
Example 31-1 Example Operations .....	1099
Example 36-1 Updating the Address Pointer Value .....	1145
Example 37-1 BS_AUX_CTRL.BO Bit Behavior .....	1158

Example 40-1 Linked-list Transfers for Memory-Mapped DMA Channels. .... 1206

# List of Tables

---

Table 1-1 ARCV2 Programming Model Configuration Options .....	69
Table 1-2 ARCV2 Instruction Set Options .....	71
Table 2-1 Single-Precision Floating-Point Data Field Descriptions.....	84
Table 2-2 Double-Precision Floating-Point Data Field Descriptions.....	85
Table 2-3 Double Precision Floating Point Data Field Descriptions .....	85
Table 2-4 Operation Description.....	92
Table 2-5 Operand Description.....	92
Table 3-1 Core Register Set .....	96
Table 3-2 16-bit Instruction Register Encoding.....	97
Table 3-3 Current ABI Register Usage.....	98
Table 3-4 Baseline Auxiliary Register Set .....	104
Table 3-5 Auxiliary Register Set .....	105
Table 3-6 Key to Auxiliary Register Access Permissions for LR and SR Instructions .....	106
Table 3-7 Zero Overhead Loop Auxiliary Registers (LP_SIZE > 0). ....	107
Table 3-8 Indexed Table Auxiliary Register (CODE_DENSITY == 1) .....	107
Table 3-9 XPU, User Extension Permission Auxiliary Register (APEX_OPTION == 1) .....	107
Table 3-10 Extension Flags Register, XFLAGS .....	108
Table 3-11 IDENTITY Field Descriptions .....	111
Table 3-12 ARCV2 Addressable Range vs. PC Size .....	113
Table 3-13 SEC_STAT Bit-Field Definitions and Read / Write Accessibility .....	114
Table 3-14 SEC_STAT Bit Debug Access.....	115
Table 3-15 SEC_STAT Field Description .....	115
Table 3-16 STATUS32 Bit-Field Definitions and Read / Write Accessibility.....	118

Table 3-17 STATUS32 Bit-Field Definitions and Read / Write Accessibility.....	119
Table 3-18 AUX_IRQ_CTRL Field Description .....	126
Table 3-19 AUX_IRQ_ACT Field Description .....	134
Table 3-20 IRQ_PRIORITY_PENDING Field Description .....	135
Table 3-21 IRQ_PRIORITY Field Description .....	137
Table 3-22 ERSEC_STAT Bit-Field Definitions and Read / Write Accessibility.....	150
Table 3-23 ERSEC_STAT Field Description .....	151
Table 3-24 AUX_SEC_EXCEPT Field Description.....	152
Table 3-25 IRQ_SELECT Field Description..	155
Table 3-26 IRQ_ENABLE Field Description .....	156
Table 3-27 IRQ_TRIGGER Field Description .....	157
Table 3-28 IRQ_STATUS Field Description .....	158
Table 3-29 IRQ_PULSE_CANCEL Field Description .....	164
Table 3-30 IRQ_PENDING Field Description.....	165
Table 3-31 Build Configuration Registers .....	167
Table 3-32 BCR_VER Field Descriptions.....	168
Table 3-33 BTA_LINK_BUILD Field Descriptions.....	169
Table 3-34 VECBASE_AC_BUILD Field Descriptions .....	170
Table 3-35 SEC_VECBASE_BUILD Field Descriptions.....	171
Table 3-36 RF_BUILD Field Descriptions .....	172
Table 3-37 MULTIPLY_BUILD Field Descriptions .....	174
Table 3-38 SWAP_BUILD Field Descriptions.....	175
Table 3-39 NORM_BUILD Field Descriptions .....	176
Table 3-40 MINMAX_BUILD field descriptions .....	177
Table 3-41 BARREL_BUILD Field Descriptions.....	178
Table 3-42 ISA_CONFIG Field Descriptions.....	179
Table 3-43 ARCH_CONFIG Field Descriptions.....	181
Table 3-44 IRQ_BUILD Field Descriptions .....	182
Table 4-1 Overview of Privileges in Kernel and User Modes .....	184

Table 4-2 Privilege Access of Instructions in the Secure Mode .....	185
Table 4-3 Interrupt Configuration Options.....	191
Table 4-4 Regular Interrupt Entry when SecureShield 2+2 Mode is Configured .....	207
Table 4-5 Regular Interrupt Exit when SecureShield 2+2 Modes is Configured.....	215
Table 4-6 Exception Vectors .....	222
Table 4-7 Exception Vectors and Cause Codes.....	226
Table 4-8 Exception and Interrupt Exit Modes.....	246
Table 5-1 Top-level Formats of the ARCv2 Instruction Set .....	252
Table 5-2 Assignment of top-level instruction formats to major opcodes in each instruction set profile .....	254
Table 5-3 Summary of 32-bit instruction formats.....	256
Table 5-4 Summary of 16-bit instruction formats.....	257
Table 5-5 Key for 32-bit Addressing Modes and Encoding Conventions .....	258
Table 5-6 Key for 16-bit Addressing Modes and Encoding Conventions .....	259
Table 5-7 Condition Codes .....	260
Table 5-8 Delay Slot Modes.....	261
Table 5-9 Encoding Static Branch Predictions for BRcc, BBIT0, and BBIT1 .....	262
Table 5-10 Address Write-Back Modes .....	262
Table 5-11 Scaling Shift .....	263
Table 5-12 Cache Bypass Modes.....	263
Table 5-13 Load Store Data Sizes .....	263
Table 5-14 Load Data Extend Mode .....	264
Table 5-15 Store Data Source Mode .....	264
Table 5-16 Instruction Syntax Convention .....	265
Table 5-17 Integer addition, subtraction and comparison operations .....	268
Table 5-18 Move operations .....	269
Table 5-19 Bit-wise logical operations .....	270
Table 5-20 Bit-mask operations.....	271
Table 5-21 Single Bit Shift and Rotate Operations .....	271
Table 5-22 Multi-bit Shift Operations.....	272

Table 5-23 Shift-assist operations . . . . .	272
Table 5-24 Selection operations . . . . .	273
Table 5-25 Byte-swapping operations . . . . .	273
Table 5-26 Bit-scanning operations . . . . .	273
Table 5-27 Relational Comparison Operations . . . . .	274
Table 5-28 Integer Multiply, MAC and Divide Operations . . . . .	274
Table 5-29 Dual and Quad Integer Multiply and MAC Operations . . . . .	275
Table 5-30 Integer Vector ADD, SUB, MPY operations . . . . .	275
Table 5-31 Special / Synchronizing Instructions . . . . .	276
Table 5-32 Memory Access Instructions . . . . .	277
Table 5-33 PREFETCH Instruction Encodings . . . . .	281
Table 5-34 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions . . . . .	282
Table 5-35 Auxiliary Register Operations . . . . .	283
Table 5-36 Control Flow Operations . . . . .	284
Table 5-37 Delay Slot Execution Modes . . . . .	285
Table 5-38 Branch on Compare/Test Mnemonics . . . . .	287
Table 5-39 Branch on Compare Pseudo Mnemonics, Register-Register . . . . .	287
Table 5-40 Branch on Compare Pseudo Mnemonics, Register-Immediate . . . . .	287
Table 5-41 Delay Slot Execution Modes . . . . .	288
Table 5-42 Special Instructions . . . . .	289
Table 5-43 Floating-Point Unit Instructions, 0x06 . . . . .	294
Table 5-44 Instruction Mnemonics . . . . .	297
Table 5-45 Basic Saturating Arithmetic Operations . . . . .	298
Table 5-46 Vector Unpacking Operations . . . . .	299
Table 5-47 Vector ALU Operations . . . . .	299
Table 5-48 Accumulator Operations . . . . .	300
Table 5-49 Vector 16x16 MAC Operations . . . . .	301
Table 5-50 Dual 16x16 MAC Operations . . . . .	302
Table 5-51 32x32 and 16x16 MAC Operations . . . . .	302

Table 5-52 Dual 16x8 MAC Operations .....	304
Table 5-53 32x16 MAC Operations .....	304
Table 5-54 Complex Operations .....	306
Table 6-1 Branch on Compare/Bit Test Register-Register .....	312
Table 6-2 Hierarchical Sub-formats of the General Operations Formats .....	317
Table 6-3 Operand Sub-format Indicators .....	318
Table 6-4 Opcode assignment for DOP instructions in major op-code 0x04 (F32_GEN4) .....	321
Table 6-5 Special DOP sub-formats in F32_GEN4 .....	322
Table 6-6 Opcode assignment for SOP instructions in major op-code 0x04 (F32_GEN4) .....	324
Table 6-7 Opcode assignment for ZOP instructions in major op-code 0x04 (F32_GEN4) .....	324
Table 6-8 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5) .....	326
Table 6-9 Opcode assignment for SOP instructions in major op-code 0x05 (F32_EXT5) .....	326
Table 6-10 Opcode assignment for ZOP instructions in major op-code 0x05 (F32_EXT5) .....	327
Table 6-11 Opcode assignment for DOP instructions in major op-code 0x06 (F32_EXT6) .....	327
Table 6-12 Opcode assignment for SOP instructions in major op-code 0x06 (F32_EXT6) .....	328
Table 6-13 Opcode assignment for ZOP instructions in major op-code 0x06 (F32_EXT6) .....	329
Table 6-14 List of FastMath extension pack instructions .....	331
Table 6-15 List of CryptoPack Extension Instructions .....	332
Table 7-1 16-Bit Indexed Jump and Link Operations .....	340
Table 7-2 16-Bit, LD / ADD Register-Register .....	341
Table 7-3 16-Bit, ADD/SUB Register-Immediate .....	342
Table 7-4 16-Bit MOV/CMP/ADD with High Register .....	344
Table 7-5 DOP_format 16-Bit General Operations .....	345
Table 7-6 16-Bit Single Operand Instructions .....	347
Table 7-7 16-Bit Zero Operand Instructions .....	348
Table 7-8 Summary of 16-Bit Load and Store Instructions with Offset .....	349
Table 7-9 16-Bit Shift/SUB/Bit Immediate .....	350
Table 7-10 Opcodes for Stack-based Operations .....	351
Table 7-11 16-Bit GP Relative Instructions .....	352

Table 7-12 16-Bit ADD/CMP Immediate . . . . .	355
Table 7-13 16-Bit Branch on Compare . . . . .	356
Table 7-14 16-Bit Branch, Branch Conditionally . . . . .	357
Table 7-15 16-Bit Branch Conditionally . . . . .	358
Table 8-1 List of Instructions . . . . .	362
Table 8-2 Delay Slot Modes for the Jump and Link Instructions . . . . .	615
Table 8-3 When Reads and Writes Take Effect . . . . .	644
Table 8-4 PREFETCH Instruction Encodings . . . . .	721
Table 8-5 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions . . . . .	722
Table 8-6 SLEEP Operand . . . . .	777
Table 9-1 Host Accesses to the ARCv2-based processor . . . . .	850
Table 9-2 Single Step Flags in Debug Register . . . . .	851
Table 11-1 Core Modes and Resource Permissions . . . . .	860
Table 11-2 ARCv2 Instruction Set Options . . . . .	862
Table 12-1 SEC_BUILD Field Description . . . . .	864
Table 13-1 ARCv2 Instruction Set Options . . . . .	868
Table 13-2 MPU Registers . . . . .	870
Table 13-3 MPU_INDEX Field Description . . . . .	873
Table 13-4 MPU_RPER Field Description . . . . .	876
Table 13-5 MPU_PROBE Field Description . . . . .	879
Table 14-1 Secure Debug Registers . . . . .	884
Table 15-1 Error Protection Auxiliary Registers . . . . .	887
Table 15-2 ERP_CTRL Field Descriptions . . . . .	888
Table 15-3 ERP_CTRL Fault Bit Descriptions . . . . .	889
Table 15-4 RFERP_STATUS_0 Field Description . . . . .	891
Table 15-5 RFERP_STATUS_1 Field Description . . . . .	893
Table 15-6 ESYN Field Description . . . . .	894
Table 15-7 ERP_BUILD Register . . . . .	895
Table 16-1 Example 1 -- Contiguous Regions Memory Map . . . . .	903

Table 16-2 Priority Configurations . . . . .	908
Table 16-3 Setting ECR Register based on EV_ProtV Exception . . . . .	909
Table 16-4 ECR parameter values for EV_ProtV sources . . . . .	910
Table 16-5 MPU Register Set . . . . .	911
Table 16-6 MPU Enable Register . . . . .	913
Table 16-7 MPU Exception Cause Register . . . . .	917
Table 16-8 MPU Region Descriptor Base Registers Field Description . . . . .	918
Table 16-9 MPU Region Descriptor Permission Registers Field Description . . . . .	919
Table 16-10 Build Configuration Registers . . . . .	920
Table 16-11 MPU_BUILD Field Description . . . . .	921
Table 17-1 Stack Checking Specification . . . . .	927
Table 17-2 Stack Region Checking Auxiliary Registers . . . . .	928
Table 17-3 STACK_REGION_BUILD Field Descriptions . . . . .	934
Table 17-4 CPROT_BUILD Field Descriptions . . . . .	938
Table 18-1 ICCM Registers . . . . .	939
Table 18-2 ICCM_BUILD Field Descriptions . . . . .	941
Table 19-1 IFQUEUE_BUILD Field Descriptions . . . . .	943
Table 20-1 Basic Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==0) . . . . .	945
Table 20-2 Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==1) . . . . .	946
Table 20-3 Instruction Cache Auxiliary Registers (IC_FEATURE_LEVEL==2) . . . . .	946
Table 20-4 IC_IVIC Description When SecureShield 2+2 mode is Configured . . . . .	947
Table 20-5 IC_CTRL Control Flags . . . . .	948
Table 20-6 IC_LIL Description . . . . .	949
Table 20-7 IC_IVIL Description . . . . .	951
Table 20-8 Cache Line Access in the Debug Unlocked in the Normal Mode . . . . .	952
Table 20-9 I_CACHE_BUILD Field Descriptions . . . . .	957
Table 21-1 DCCM Auxiliary Registers . . . . .	959
Table 21-2 DCCM_BUILD Field Descriptions . . . . .	961
Table 22-1 Basic Data Cache Auxiliary Registers (DC_FEATURE_LEVEL==0) . . . . .	963

Table 22-2 Data Cache Lock/Flush Registers (DC_FEATURE_LEVEL > 0) .....	963
Table 22-3 Data Cache Auxiliary Registers for Direct RAM Access (DC_FEATURE_LEVEL >1) .....	964
Table 22-4 DC_IVDC Field Descriptions.....	965
Table 22-5 DC_IVDC Description When SecureShield 2+2 Option is Configured.....	966
Table 22-6 DC_CTRL Control Flags .....	967
Table 22-7 DC_LDL Description When SecureShield 2+2 mode Option is Configured .....	969
Table 22-8 DC_IVDL Description When SecureShield 2+2 mode Option is Configured .....	972
Table 22-9 DC_FLSH Control Flags .....	973
Table 22-10 DC_FLSH Description When ESP is Configured .....	974
Table 22-11 DC_FLDL Description When SecureShield 2+2 mode is Configured .....	976
Table 22-12 Cache Line Access in the Debug Unlocked in the Normal Mode .....	977
Table 22-13 DC_TAG Control Flags and Update Conditions .....	978
Table 22-14 Build Configuration Registers .....	983
Table 22-15 D_CACHE_BUILD Field Descriptions.....	984
Table 23-1 NV_ICCM Registers .....	987
Table 23-2 NV_ICCM_CTRL Field Descriptions .....	988
Table 23-3 NV_ICCM_ERASE Field Descriptions .....	990
Table 24-1 Calibration Storage Registers.....	993
Table 24-2 KEY_STORE_CTRL Field Descriptions .....	995
Table 24-3 KEY_STORE_PWD Field Descriptions.....	997
Table 24-4 KEY_STORE_BUILD.....	1001
Table 25-1 Calibration Storage Registers.....	1003
Table 25-2 CAL_STORE_CTRL Field Descriptions .....	1004
Table 25-3 CAL_STORE_BUILD.....	1008
Table 26-1 Auxiliary Registers for Hardware Counter 0 (has_timer_0 == true) .....	1010
Table 26-2 Auxiliary Registers for Hardware Counter 1 (has_timer_1 == true) .....	1010
Table 26-3 Auxiliary Register for Real-time counter .....	1011
Table 26-4 Auxiliary Registers for Hardware Counter 0 (-sec_timer_0==true) .....	1011
Table 26-5 Auxiliary Registers for Hardware Counter 0 (-sec_timer_1==true) .....	1011

Table 26-6 RTC Control Register .....	1024
Table 26-7 TIMER_BUILD Field Descriptions .....	1028
Table 26-8 Error Protection Auxiliary Registers.....	1029
Table 26-9 WDT_PASSWD Field Descriptions.....	1030
Table 26-10 WDT_CTRL Field Descriptions.....	1031
Table 27-1 Safety Island Registers .....	1037
Table 27-2 Enabling Lockstep Modes .....	1038
Table 27-3 LSC_CTRL Register.....	1038
Table 27-4 LSC_MCD Field Description .....	1040
Table 27-5 LSC_STATUS Field Description .....	1041
Table 27-6 LSC_ERROR_REG Field Description .....	1042
Table 27-7 LSC_BUILD_AUX Field Description .....	1044
Table 28-1 DMP Registers .....	1045
Table 29-1 Host Debug Interface Auxiliary Registers .....	1047
Table 29-2 DEBUG Register Field Description.....	1048
Table 29-3 Internal Clock During Sleep Mode .....	1051
Table 29-4 DEBUGI Register Field Description .....	1052
Table 29-5 ACG_CTRL Field Description .....	1054
Table 30-1 Auxiliary Registers for Actionpoints .....	1056
Table 30-2 State of Auxiliary Registers upon Reset .....	1058
Table 30-3 Actionpoint Build Field Descriptions.....	1060
Table 30-4 Actionpoint Target Field, AT, Type .....	1062
Table 30-5 Transaction Type Field, TT, Type .....	1063
Table 30-6 Mode Field, M, Type .....	1063
Table 30-7 Pair Field, P, Type .....	1064
Table 30-8 Actionpoint Action Field, AA Field, Type .....	1065
Table 30-9 Quad Type .....	1066
Table 30-10 SMART_BUILD .....	1076
Table 30-11 SMART_CONTROL Register.....	1077

Table 30-12 SMART_DATA Register .....	1078
Table 30-13 SRC_ADDR Value .....	1078
Table 30-14 DEST_ADDR Register .....	1079
Table 30-15 FLAGS Value Register .....	1079
Table 31-1 Performance Auxiliary Registers .....	1082
Table 31-2 Countable-Conditions Registers .....	1082
Table 31-3 CC_INDEX Field Description .....	1084
Table 31-4 CC_NAME0 Field Description .....	1085
Table 31-5 CC_NAME1 Field Description .....	1086
Table 31-6 PCT_CONFIG Field Description .....	1091
Table 31-7 PCT_CONTROL Field Description .....	1092
Table 31-8 PCT_INDEX Field Description .....	1094
Table 31-9 PCT_UFLAGS Field Description .....	1100
Table 31-10 PCT_BUILD Field Description .....	1101
Table 31-11 CC_BUILD Field Description .....	1102
Table 32-1 ARC RTT Registers .....	1103
Table 32-2 RTT_COMMAND Bit Field Description .....	1106
Table 32-3 RTT_BUILD Field Descriptions .....	1107
Table 33-1 Power Domain Management Registers .....	1109
Table 33-2 PDM_PSTAT Field Description .....	1110
Table 33-3 RTT_PDM_PSTAT Field Description .....	1112
Table 33-4 DVFS_PL Field Description .....	1113
Table 33-5 PDM_PMODE Field Description .....	1114
Table 33-6 PDM_DVFS_BUILD Field Description .....	1115
Table 34-1 FPU Registers .....	1119
Table 34-2 FPU_CTRL Field Description .....	1120
Table 34-3 FPU_STATUS Field Description .....	1121
Table 34-4 AUX_DPFP1L Field Description .....	1124
Table 34-5 AUX_DPFP1H Field Description .....	1125

Table 34-6 AUX_DPFP2L Field Description .....	1126
Table 34-7 AUX_DPFP2H Field Description .....	1127
Table 34-8 FPU_BUILD Field Descriptions .....	1128
Table 35-1 XY Registers .....	1134
Table 35-2 XY_BUILD Field Descriptions .....	1137
Table 36-1 AGU Registers .....	1141
Table 36-2 Number of Address Pointer Registers .....	1142
Table 36-3 Post-Incrementing Address Pointer Modes .....	1143
Table 36-4 AGU_AUX_APx Field Description .....	1144
Table 36-5 Number of Offset or Wrap Registers .....	1147
Table 36-6 Number of Modifier Registers .....	1148
Table 36-7 Data Transformation-Related Fields in the Modifier Registers .....	1152
Table 36-8 Data Transformation Modes Based on the FX Field .....	1152
Table 36-9 AGU_BUILD Field Descriptions .....	1155
Table 37-1 Bitstream Auxiliary Registers .....	1157
Table 37-2 BS_AUX_CTRL Field Description .....	1158
Table 37-3 BS_AUX_ADDR Field Description .....	1159
Table 37-4 BS_AUX_BIT_OS Field Description .....	1160
Table 37-5 BS_AUX_WDATA Field Description .....	1161
Table 37-6 Build Configuration Registers .....	1161
Table 37-7 BS_BUILD Field Descriptions .....	1162
Table 39-1 ARConnect Registers .....	1165
Table 39-2 MCIP_CMD Register Field Description .....	1167
Table 39-3 MCIP_SYSTEM_BUILD Register Field Description .....	1170
Table 39-4 MCIP_SEMA_BUILD Register Field Description .....	1172
Table 39-5 MCIP_MESSAGE_BUILD Register Field Description .....	1173
Table 39-6 CONNECT_PMU_BUILD Field Description .....	1175
Table 39-7 MCIP_GFRC_BUILD Field Description .....	1175
Table 39-8 MCIP_ICI_BUILD Field Description .....	1176

Table 39-9 MCIP_ICD_BUILD Field Description . . . . .	1177
Table 39-10 MCIP_PDM_BUILD Field Description . . . . .	1178
Table 40-1 DMA Auxiliary Registers . . . . .	1181
Table 40-2 XY DMA Controller Auxiliary Registers . . . . .	1183
Table 40-3 DMACTRL Field Description . . . . .	1184
Table 40-4 DMACENB Field Description . . . . .	1185
Table 40-5 DMACTS Field Description . . . . .	1186
Table 40-6 DMAHPRI Field Description . . . . .	1187
Table 40-7 DMACNPRI Field Description . . . . .	1188
Table 40-8 DMACREQ Field Description . . . . .	1189
Table 40-9 DMACTSTAT0 Field Description . . . . .	1190
Table 40-10 DMACTSTAT1 Field Description . . . . .	1191
Table 40-11 DMAIRQ Field Description . . . . .	1193
Table 40-12 Memory-Mapped DMA Channel-Specific Registers . . . . .	1195
Table 40-13 DMA Channel Auxiliary Registers . . . . .	1196
Table 40-14 DMA Channel Registers . . . . .	1196
Table 40-15 DMACTRLx OP Field Description . . . . .	1197
Table 40-16 DMACTRLx RT Field Description . . . . .	1198
Table 40-17 DMACTRLx DTT Field Description . . . . .	1199
Table 40-18 DMACTRLx DW Field Description . . . . .	1199
Table 40-19 DMACTRLx BLOCK_SIZE Field Description . . . . .	1200
Table 40-20 DMACTRLx ARB Field Description . . . . .	1200
Table 40-21 DMACTRLx I Bit Description . . . . .	1201
Table 40-22 DMACTRLx AM Field Description . . . . .	1201
Table 40-23 DMA Channel Descriptors . . . . .	1206
Table 40-24 Initial State of DMA Channel Descriptor Registers for Example 40-1 . . . . .	1206
Table 40-25 DMA Channel Descriptors . . . . .	1207
Table 40-26 Final State of DMA Channel Descriptor Registers for Example 40-2 . . . . .	1207
Table 40-27 DMA BUILD Field Descriptions . . . . .	1210

Table 41-1 ARCV2 Programming Model Configuration Options .....	1217
Table 41-2 ARCV2 Instruction Set Options .....	1218
Table 42-1 8-Bit Signed Integer Vector .....	1219
Table 42-2 16-Bit Signed Integer .....	1219
Table 42-3 16-Bit Unsigned Integer .....	1220
Table 42-4 16-Bit Signed Integer Vector .....	1220
Table 42-5 16-Bit Unsigned Integer Vector .....	1220
Table 42-6 16-Bit Signed Fractional Vector .....	1221
Table 42-7 Complex 16+16 Bit Complex Signed Fractional Data .....	1221
Table 42-8 32-Bit Signed Integer .....	1221
Table 42-9 32-Bit Unsigned Integer .....	1221
Table 42-10 32-Bit Signed Fractional Data .....	1222
Table 42-11 Accumulator Instructions .....	1224
Table 43-1 DSP Auxiliary Registers .....	1228
Table 43-2 ACC0_GLO Field Descriptions .....	1230
Table 43-3 ACC0_GHI Field Descriptions .....	1232
Table 43-4 DSP_FFT_CTRL Field Descriptions .....	1234
Table 43-5 DSP_CTRL Field Descriptions .....	1235
Table 43-6 DSP_BUILD Field Descriptions .....	1237
Table 44-1 List of Instructions .....	1239
Table A-1 Auxiliary Register List .....	1705
Table A-2 Build Configuration Registers .....	1712
Table B-1 Baseline Differences: ARC 600, ARC 700, and ARCV2 .....	1716
Table B-2 ISA Option Differences: ARC 600, ARC 700, and ARCV2 .....	1718
Table B-3 ISA Features: ARC EM and ARC HS .....	1720
Table C-1 Implementation behavior on division overflow .....	1724



# Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

## Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services, which include downloading software, viewing Documentation on the Web, and entering a call to the Support Center.

To access SolvNet:

1. Go to the SolvNet Web page at <http://solvnet.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 SolvNet.)

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

## Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to <http://solvnet.synopsys.com/> (Synopsys user name and password required), clicking "Enter a Call to the Support Center."
- Send an e-mail message to your local support center.
  - E-mail [support\\_center@synopsys.com](mailto:support_center@synopsys.com) from within North America.
  - Find other local support center e-mail addresses at [http://www.synopsys.com/support/support\\_ctr](http://www.synopsys.com/support/support_ctr).
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at [http://www.synopsys.com/support/support\\_ctr](http://www.synopsys.com/support/support_ctr).

## 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
5755-080	December 2016	ARCv2 ISA updates for ARC EM 4.0 and ARC SEM 1.0 family of processors. Added support for the following secure features: <ul style="list-style-type: none"><li>■ SecureShield 2+2 modes</li><li>■ Secure MPU</li><li>■ Side-channel protection</li><li>■ ARCv2 ISA alignment</li><li>■ Updates to the XY AGU modifiers</li></ul>
5755-073	February 2016	ARCv2 ISA updates for ARC EM 3.1 family of processors
5755-072	September 2015	ARCv2 ISA updates for ARC EM 3.0 family of processors
5755-046	November 2014	<ul style="list-style-type: none"><li>■ Added the DEBUG.EH bit</li></ul>
5755-035	August 2014	ARCV2 ISA for ARC EM 2.0 family of processors: <ul style="list-style-type: none"><li>■ ARCV2DSP (ARCV2ISA extension) with more than 100 DSP instructions</li><li>■ Added DSP support</li><li>■ Added FPU support</li><li>■ Added performance monitor support</li><li>■ Added ARC RTT (ARC Real-time Trace) support</li><li>■ Added ARConnect (multi-core IP) support</li></ul>
5755-011	October 2012	ARCV2 ISA release for ARC EM family of processors
5765-002	December 2013	Added descriptions for the following registers: <ul style="list-style-type: none"><li>■ Watchdog Timer registers<ul style="list-style-type: none"><li>□ WDT_CTRL</li><li>□ WDT_PASWD</li><li>□ WDT_COUNT</li><li>□ WDT_PERIOD</li></ul></li><li>■ SEP_CTRL and SEP_BUILD registers</li><li>■ Updated first technical review comments from R&amp;D</li></ul>
5765-001	October 2013	Initial version

# 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](#)



# 1

# 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.

## 1.1 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.

## 1.2 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

## 1.3 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.

### 1.3.1 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-r28, r30 and r31) 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 [95](#).

Additional banks of registers may be configured for fast interrupt response and context switching. See "[Banked Interrupt Registers](#)" on page [195](#).

### 1.3.2 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 (i.e. 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 [104](#).

### 1.3.3 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, LIMM \(r62\)](#)) 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 64-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.

### 1.3.4 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 ±16MB to maximum offset of ±512B
- 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](#)))

### 1.3.5 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](#).

## 1.4 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 1-1** describes the configuration options for the programming model supported by the ARCv2 architecture.



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

---

### 1.4.1 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 1-1](#).

**Table 1-1 ARCV2 Programming Model Configuration Options**

Configuration Option	Value Range	Description
-byte_order	LITTLE_ENDIAN, BIG_ENDIAN	Defines the ordering of bytes within a multi-byte memory word.
-addr_size	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.
-pc_size	16, 20, 24, 28,32 bits	Determines the maximum addressable code space. This option impacts PC-related hardware. This value cannot exceed -addr_size, but can be smaller than or the same as -addr_size.
-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.
-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. <ul style="list-style-type: none"> <li>■ false: indicates that the interrupt unit is not included.</li> <li>■ true: indicates that the interrupt unit is included.</li> </ul>
-halt_on_reset	false, true	Indicates whether the core is halted initially on reset. <ul style="list-style-type: none"> <li>■ false: core is not halted initially on reset.</li> <li>■ true: core is halted initially on reset.</li> </ul>

**Table 1-1 ARCv2 Programming Model Configuration Options (Continued)**

Configuration Option	Value Range	Description
-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	16 (0 to 15 interrupt priorities)	Defines the number of interrupt priority levels 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.

## 1.4.2 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 *Appendix: Processor Configuration Options*.

### 1.4.3 ISA Options

Table 1-2 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 361.

**Table 1-2 ARCv2 Instruction Set Options**

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-code_density_option <sup>a</sup>	0	-	-
	1	SETEQ, SETNE, SETLT, SETGE, SETLO, SETHS, SETLE, SETGT ENTER_S, LEAVE_S, BI, BIH LDI, LDI_S	LDI_BASE
	2	As CODE_DENSITY_OPTION 1, plus LD_S R0-3,[h,u5] LD_S.AS a,[b,c] LD_S R1,[GP, s11] ST_S R0,[GP,s11]	
	3	As CODE_DENSITY_OPTION 2, plus JLI_S EI_S	JLI_BASE EI_BASE
-bitscan_option	false	-	-
	true	NORM, NORMH, FFS, FLS	-
-swap_option	false	-	-
	true	SWAP, SWAPE, LSL16, LSR16	-

**Table 1-2** 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 ARCVER value of 0x50 or above.	
-os_opt_option	0	-	
	1	PREALLOC, SWI_S encoding with n6 operand, DSYNC, DMB	
-mpy_option	none	-	-
	wlh1	MPYW, MPYUW	-
	wlh2 to wlh5	All mpy_option=wlh1 instructions and MPY, MPYU, MPYM, MPYMU, MPY_S,	-
	plus_macd	All mpy_option=plus_dmpy instructions and the following: MPYD, MPYDU, MACD, MACDU, VMPY2H, VMPY2HU, VMAC2H, VMAC2HU	

**Table 1-2** ARCv2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
	plus_qmacw	All mpy_option=plus_macd instructions and the following: QMPYH, QMPYHU, DMPYW, DMPYWH, QMACH, QMACHU, DMACWH, DMACWHU, VADD4H, VSUB4H, VADDSUB4H, VSUBADD4H, VADD2, VSUB2, VADDSUB, VSUBADD	
-atomic_option	false		-
	true	LLOCK, SCOND, WLFC LLOCKD and SCOND instructions also require -ll64_option==true	-
-div_rem_option	none	-	DZ bit in STATUS32
	radix4_enhanced	DIV, DIVU, REM, REMU	
-ll64_option	false		
	true	LD, STD LLOCKD and SCOND instructions also require -atomic_option==true	
-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	

**Table 1-2** ARCv2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-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	1	DMULH11, DMULH12, DMULH21, DMULH22, DADDH11, DADDH12, DADDH21, DADDH22, DSUBH11, DSUBH12, DSUBH21, DSUBH22, DEXCL1, DEXCL2	
-has_dsp	1	See <a href="#">Table 44-1</a>	
-dsp_complex	1	Instructions included when HAS_DSP==1 and instructions listed in <a href="#">Table 5-54</a> on page 306.	
-sec_modes_option	false	-	
	true	SJLI, SFLAG	<ul style="list-style-type: none"> <li>■ NSC_TABLE_TOP</li> <li>■ NSC_TABLE_BASE</li> <li>■ AUX_KERNEL_SP</li> <li>■ AUX_S_USER_SP</li> <li>■ AUX_S_KERNEL_SP</li> <li>■ AUX_SEC_EXCEPT</li> <li>■ ERSEC_STAT</li> <li>■ SEC_STAT</li> <li>■ SEC_VECBASE_BUILD</li> <li>■ 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 <a href="#">SecureShield™ Technology</a>.</li> </ul>

**Table 1-2** ARCv2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
- intvbase_preset_s	PC range		<p>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.</p> <p>Note: This option is available only when -sec_modes_option==true.</p>

- a. Some implementations may not distinguish between the intermediate levels of CODE\_DENSITY\_OPTION, offering the entire set of code density options as a single “all or nothing” option. Please refer to your processor’s Databook for further information on the interpretation of CODE\_DENSITY\_OPTION on your ARC processor.

## 1.5 Custom Extensions

The ARCv2-based processor is designed to be extendable 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.

### 1.5.1 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 3-1](#) on page [95](#) shows the core register map.

### 1.5.2 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 basemode 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.

### 1.5.3 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.

### 1.5.4 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 basemode 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 5-7](#) on page [260](#).

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



## 2

# Data Organization and Addressing

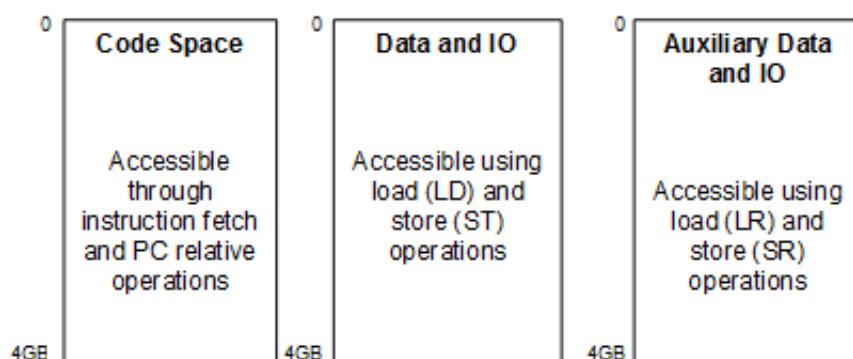
This chapter describes the data organization and addressing used by the ARCv2-based processor.

## 2.1 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 2-1](#) and [2.2](#) for more information.

**Figure 2-1 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.

## 2.2 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 2-5](#) and [Figure 2-6](#).

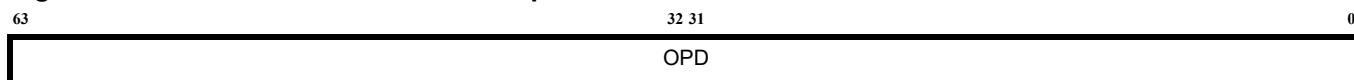
### 2.2.1 Vector Operands

Vector instructions (such as QMACH) 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.

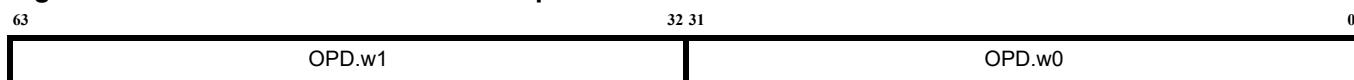
#### 2.2.1.1 64-Bit Operand Data Elements

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.

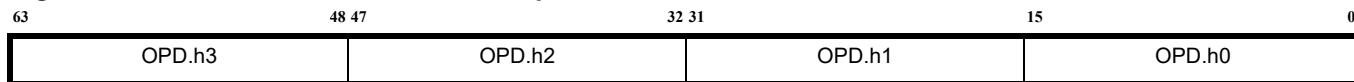
**Figure 2-2 64-bit Element As a 64-bit Operand**



**Figure 2-3 32-bit Elements in a 64-bit Operand**



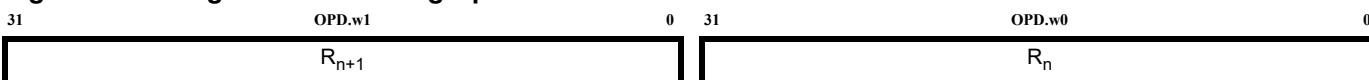
**Figure 2-4 16-bit Elements in a 64-bit Operand**

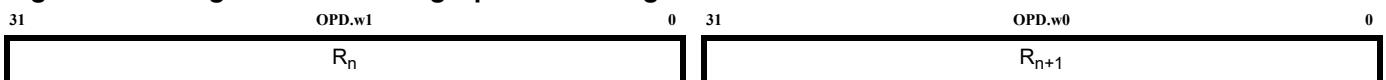


### Double-Word Operations

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 2-5 Register Pair Holding Operands in Little-endian Mode**



**Figure 2-6 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.

### 2.2.1.2 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 hence 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  $R_n$ , [A]

Little-endian mode (LE)

$$\begin{aligned} R_{n+1} &\leftarrow \{ A+7, A+6, A+5, A+4 \} \\ R_n &\leftarrow \{ A+3, A+2, A+1, A \} \end{aligned}$$

Big-endian mode (BE)

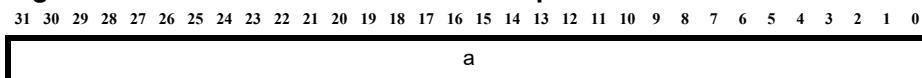
$$\begin{aligned} R_{n+1} &\leftarrow \{ A+4, A+5, A+6, A+7 \} \\ R_n &\leftarrow \{ A, A+1, A+2, A+3 \} \end{aligned}$$

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.

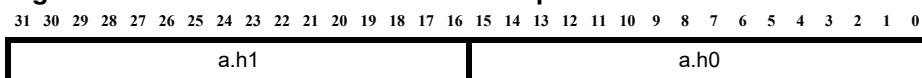
If a 64-bit object has been loaded from memory, the most-significant word is in  $R_n$  for a big-endian system, or  $R_{n+1}$  for a little-endian system. Likewise, the least-significant word is in  $R_{n+1}$  for a big-endian system, or  $R_n$  for a little-endian system. Hence, instructions that operate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

## 2.2.2 32-Bit Operand Data Elements

**Figure 2-7** 32-Bit Elements in a 32-bit Operand



**Figure 2-8** 16-Bit Elements in a 32-bit Operand

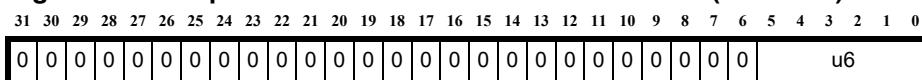


## 2.2.3 Expansion of Literals

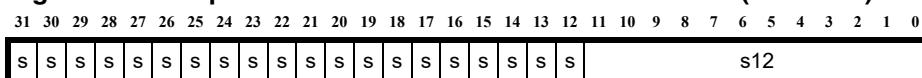
The u6 and s12 instruction literals are referred to as short-immediate (or shimm) operands. When short-immediate (shimm) or long-immediate (limm) operands are used in a 64-bit context (except STD) or in vector arithmetic instructions, the following diagrams illustrate the expansion of the literals to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

[Figure 2-9](#) and [Figure 2-10](#) illustrate expansion of a shimm to a word (32 bits).

**Figure 2-9 Expansion of a u6 Literal to a 32-Bit Word (w-shimm)**

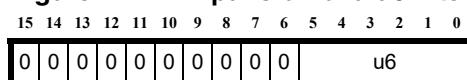


**Figure 2-10 Expansion of an s12 Literal to a 32-Bit Word (w-shimm)**

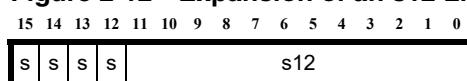


**Figure 2-11** and **Figure 2-12** illustrate expansion of a shimm to a half-word (16-bits).

**Figure 2-11 Expansion of a u6 Literal to Half-Word (16-bit, hw-shimm)**



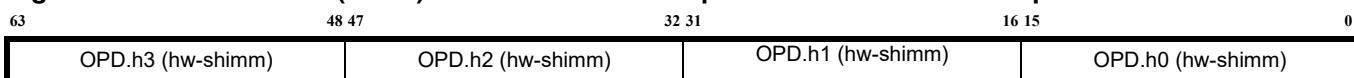
**Figure 2-12 Expansion of an s12 Literal to Half-Word (16-bit, hw-shimm)**



#### 2.2.4 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 below:

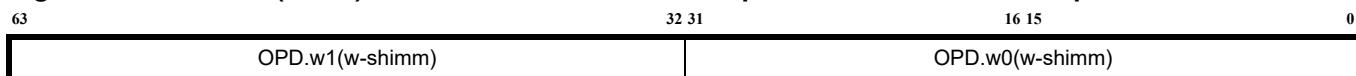
**Figure 2-13 A Half-Word (16-bit) Short Limm Value Replicated to form a 64-Bit Operand**



Where a hw-shimm represents u6 or s12 extended to half-word (16-bits). For an illustration of hw-shimm, see [Figure 2-11](#) and [Figure 2-12](#).

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

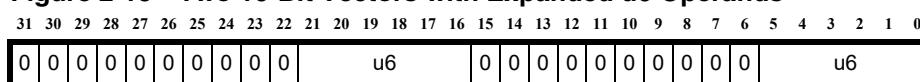
**Figure 2-14** A word (32-bit) Short-limm or Limm value Replicated to form a 64-Bit Operand



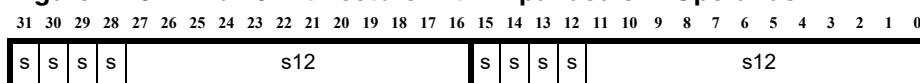
Where a w-shimm represents u6 or s12 expanded to a word (32-bits), or a limm. For an illustration of w-shimm, see [Figure 2-9](#) and [Figure 2-10](#).

**Figure 2-15** and **Figure 2-16** illustrate the expansion of two hw-shimm(16 bits) to a word (32 bits).

**Figure 2-15** Two 16-Bit Vectors with Expanded u6 Operands

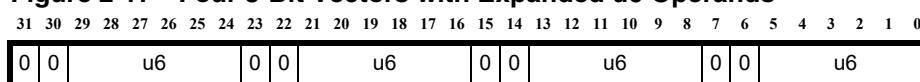


**Figure 2-16** Two 16-Bit Vectors with Expanded s12 Operands

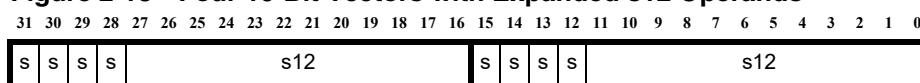


[Figure 2-17](#) and [Figure 2-18](#) illustrate the expansion of four 8-bit vectors (expanded u6 or s12 literals) to a word (32 bits).

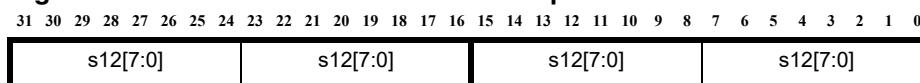
**Figure 2-17** Four 8-Bit Vectors with Expanded  $y_6$  Operands



**Figure 2-18 Four 16-Bit Vectors with Expanded s12 Operands**



**Figure 2-19** Four 8-Bit Vectors with s12 Operands



### 2.2.5 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 2-20 illustrates the expansion of a w6 in STD.

**Figure 2-20 Expansion of a w6 in STD**

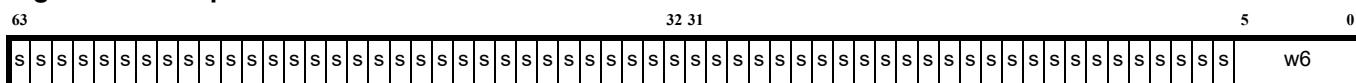
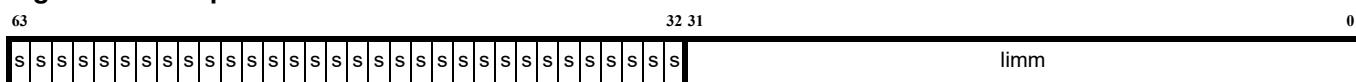


Figure 2-21 illustrates the expansion of a limm in STD.

**Figure 2-21 Expansion of a limm in STD**



## 2.3 Floating-Point Data Formats

The floating-point unit supports the following data formats:

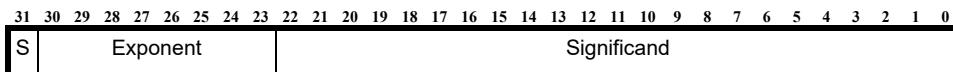
- “[Single-Precision Floating-Point Data Formats](#)” on page [84](#)
- “[Double-Precision Floating-Point Data Formats](#)” on page [84](#)
- “[NaN Formats](#)” on page [85](#)

### 2.3.1 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 2-22](#).

**Figure 2-22 Single-Precision Floating-Point Data**



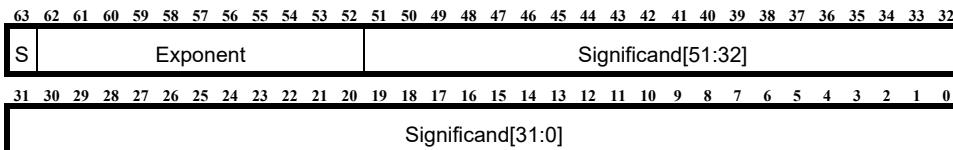
**Table 2-1 Single-Precision Floating-Point Data Field Descriptions**

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

### 2.3.2 Double-Precision Floating-Point Data Formats

When hardware support for double-precision floating-point operations is configured (i.e. when `FPU_DP_OPTION == 1`), a double-precision floating-point number is represented in core registers as shown in [Figure 2-23](#).

**Figure 2-23 Double-Precision Floating-Point Data**



The bit fields are described in the following table.

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

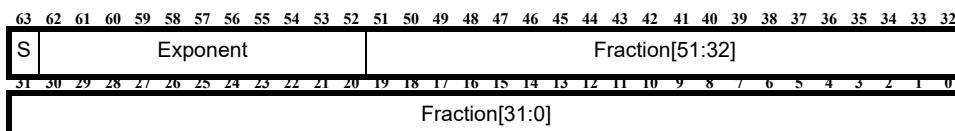
Field	Description
S	Sign part of the floating-point number 0: Positive Number 1: Negative Number
Exponent	Exponent part of the floating-point number
Significand	Significand part of the floating-point number

## 2.4 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 2-24](#).

**Figure 2-24 Double Precision Floating Point Data****Table 2-3 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

### 2.4.1 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



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

0 \* infinity

or

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.

## 2.5 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.

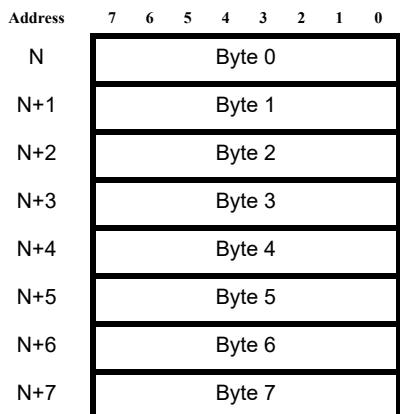
Software can detect whether non-aligned memory accesses are supported by examining the N (NON\_ALIGNED) field of the [Instruction Set Configuration Register, ISA\\_CONFIG](#) build configuration register. When this field is set to 1, the implementation supports non-aligned memory accesses.

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.

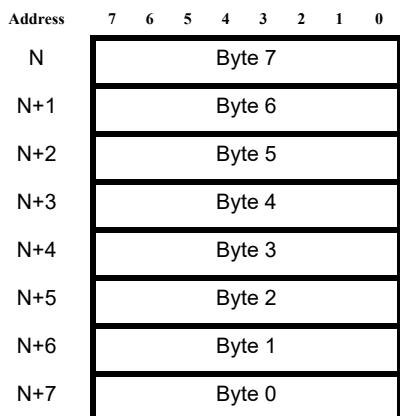
## 2.5.1 64-Bit Data

[Figure 2-25](#) 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 2-26](#)

**Figure 2-25 64-bit Register Data in Byte-Wide Memory, Little-Endian**



**Figure 2-26 64-bit Register Data in Byte-Wide Memory, Big-Endian**

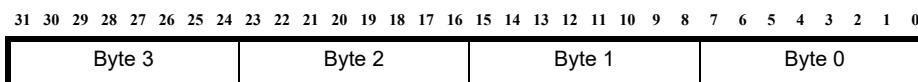


For information about the 64-bit data organization in a register pair, see [Figure 2-5](#) and [Figure 2-6](#).

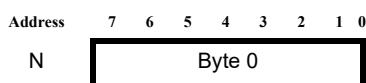
## 2.5.2 32-Bit Data

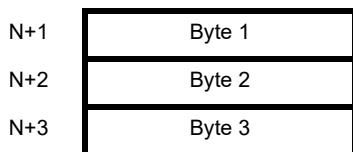
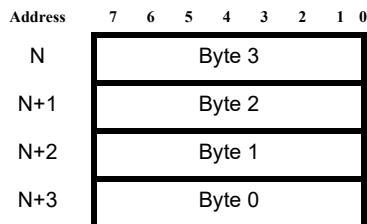
All load or store, arithmetic, and logical operations support 32-bit data. [Figure 2-27](#) on page 88 show the data representation in a general purpose register. [Figure 2-28](#) on page 88 shows the little-endian representation in byte-wide memory. [Figure 2-29](#) on page 89 shows the big-endian representation.

**Figure 2-27 Register containing 32-bit Data**



**Figure 2-28 32-bit Register Data in Byte-Wide Memory, Little-Endian**

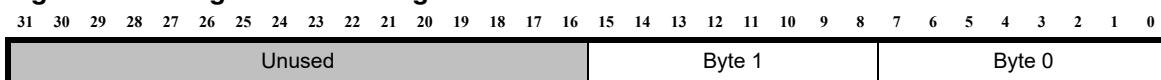
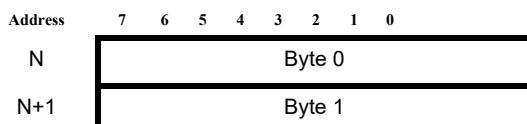
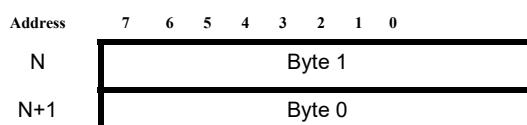


**Figure 2-28 32-bit Register Data in Byte-Wide Memory, Little-Endian (Continued)****Figure 2-29 32-bit Register Data in Byte-Wide Memory, Big-Endian**

### 2.5.3 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 2-30](#) on page 89 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 2-31](#) on page 89 shows the little-endian representation of 16-bit data in byte-wide memory. If the ARCv2-based processor supports big-endian addressing, [Figure 2-32](#) on page 89 shows how the 16-bit data is stored in memory.

**Figure 2-30 Register Containing 16-Bit Data****Figure 2-31 16-bit Register Data in Byte-Wide Memory, Little-Endian****Figure 2-32 16-bit Register Data in Byte-Wide Memory, Big-Endian**

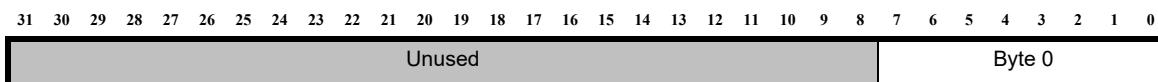
## 2.5.4 8-Bit Data

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 (SEXBT) instructions. [Figure 2-33](#) on page 90 shows the 8-bit data representation in a general purpose register.

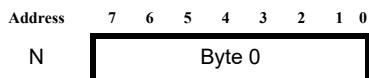
For the programmer's model, the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a *data invariance* principle. [Figure 2-34](#) on page 90 shows the representation of 8-bit data in byte-wide memory.

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 2-33 Register Containing 8-Bit Data**



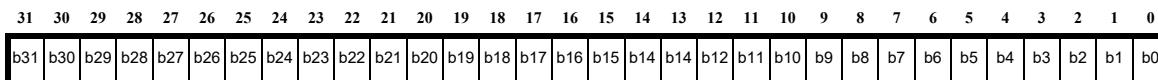
**Figure 2-34 8-bit Register Data in Byte-Wide Memory**



## 2.5.5 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 2-35 Register Containing 1-Bit Data**

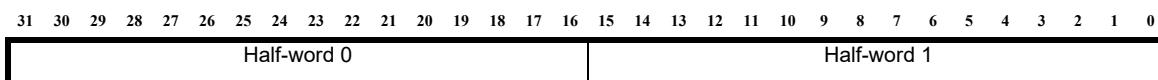


## 2.6 Instruction Layout in Memory

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 2-36 Half-word Numbering of a 32-Bit Instruction or Long-immediate Data Item**



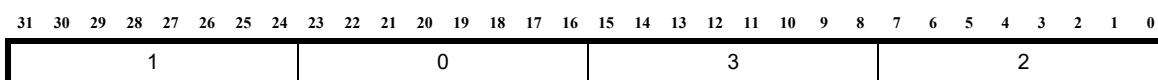
[Figure 2-36](#) shows the number of two half-words in a 32-bit instruction or long-immediate data item.

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

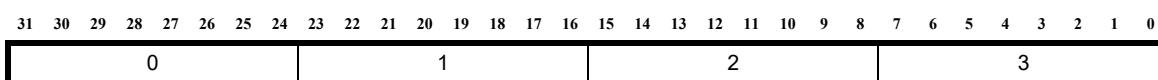
The location of each byte of an instruction or long-immediate value depends on the memory byte ordering configured for the processor. An AR Cv2-based processor can be configured to support either big-endian or little-endian byte ordering. This configuration is always static, and cannot be altered at runtime.

[Figure 2-37](#) 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 2-38](#) shows the corresponding big-endian addresses.

**Figure 2-37 Little-endian Offset of each Byte in a 32-Bit Instruction or Immediate**

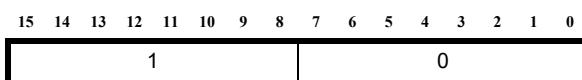


**Figure 2-38 Big-endian Offset of each Byte in a 32-Bit Instruction or Immediate**

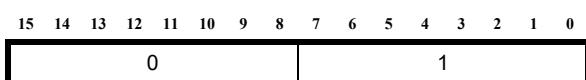


Similarly, Figure 2-39 illustrates the little-endian memory address offset of each byte within a 16-bit instruction, relative to the start address of that instruction. Figure 2-40 illustrates the big-endian memory offset of each byte in a 16-bit instruction.

**Figure 2-39 Little-endian Memory Offset of each Byte in a 16-Bit Instruction**



**Figure 2-40 Big-endian Memory Offset of each Byte in a 16-bit Instruction**



## 2.6.1 Addressing Modes

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

<b>Register Direct</b>	Operations are performed on values stored in the registers
<b>Register Indirect</b>	Operations are performed on locations specified by the contents of registers.
<b>Register Indirect with offset</b>	Operations are performed on locations specified by the contents of a register and an offset value (in another register, or as immediate data)
<b>Immediate</b>	Operations are performed by using the constant data stored within the opcode.
<b>PC relative</b>	Operations are performed relative to the current value of the Program Counter (usually branch or PC relative loads).
<b>Absolute</b>	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 2-4 Operation Description**

Operation	
<.f>	writeback 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, Word, or Long)
<.x>	perform sign extension
<.di>	data cache bypass (load and store operations)
<.aa>	address writeback
<.T>	invert the default static branch prediction (see “ <a href="#">Static Branch Prediction Mode</a> ” on page <a href="#">312</a> )

**Table 2-5 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.

**Table 2-5    Operand Description**

Operand	
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)

## 2.6.2    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 2-1    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.

## 2.6.3    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.

## 2.6.4    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.

## 2.6.5 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' (B<sub>BIT</sub>0, B<sub>BIT</sub>1) 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.

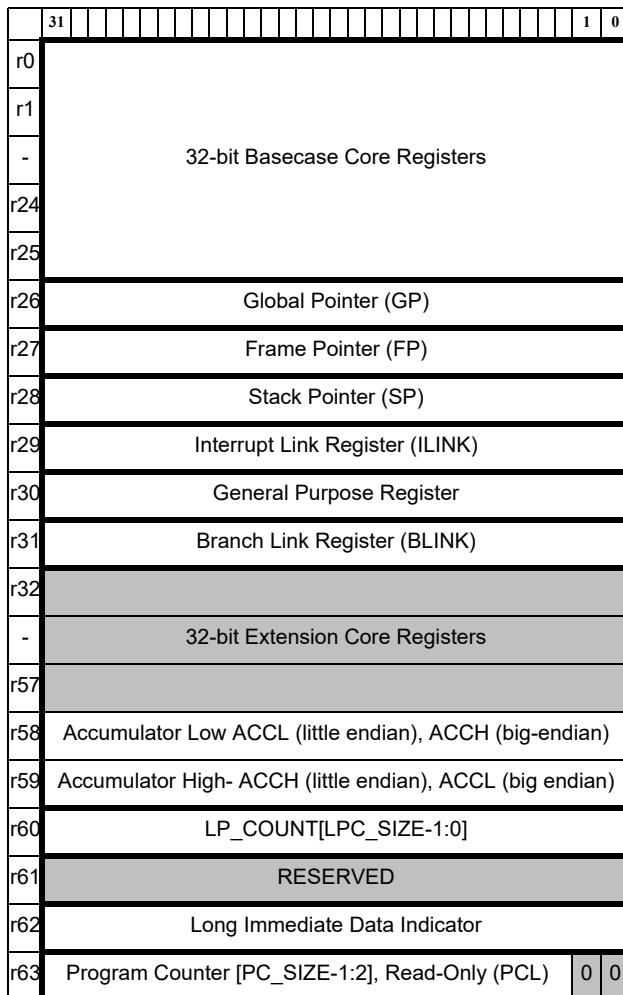
3

# Core Architectural State Registers

### 3.1 Core Register Set

The AR Cv2 programming model provides a set of general-purpose registers. Figure 3-1 shows a summary of the core register set.

### **Figure 3-1 Core Register Map Summary**



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, in order 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 3-1](#) on page [96](#) lists summarizes the accessibility of core registers from 16-bit and 32-bit instruction formats.

**Table 3-1 Core Register Set**

<b>Register</b>	<b>32-bit Instruction Function and Default Usage</b>	<b>16-bit Instruction Access to Register</b>
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 <a href="#">Pointer Registers, GP (r26), FP (r27), SP (r28)</a> )	Global pointer	LD_S, MOV_S, CMP_S, ADD_S
r27 (FP) (see <a href="#">Pointer Registers, GP (r26), FP (r27), SP (r28)</a> )	Frame pointer (default)	MOV_S, CMP_S, ADD_S
r28 (SP) (see <a href="#">Pointer Registers, GP (r26), FP (r27), SP (r28)</a> )	Stack pointer	PUSH_S, POP_S, SUB_S, LD_S, ST_S, MOV_S, CMP_S, ADD_S
r29 (ILINK) (see <a href="#">Link Registers, ILINK (r29), BLINK (r31)</a> )	Interrupt link	
r30	General purpose	Default access. Long immediate indicator for instructions with 'h' operand
r31 (BLINK) (see <a href="#">Link Registers, ILINK (r29), BLINK (r31)</a> )	Branch link register	JL_S, BL_S, J_S, PUSH_S, POP_S, MOV_S, CMP_S, ADD_S

**Table 3-1 Core Register Set (Continued)**

Register	32-bit Instruction Function and Default Usage	16-bit Instruction Access to Register
r32 - r57	Extension core registers	
r58	Accumulator	
r59	Accumulator	
r60 (LP_COUNT) (see <a href="#">Loop Count Register, LP_COUNT (r60)</a> )	Loop counter	
r61	Reserved	Reserved
R62	Long immediate	MOV_S, CMP_S, ADD_S
R63 (PCL) (see <a href="#">Word-aligned Program Counter, PCL (r63)</a> )	Program counter (32-bit aligned)	

### 3.1.1 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 3-2](#) on page 97 lists the special register encoding and [Table 3-3](#) on page 98 lists the ABI usage support.

**Table 3-2 16-bit Instruction Register Encoding**

16-bit Instruction Register Encoding	32-bit Instruction Register
0	r0
1	r1
2	r2
3	r3
4	r12
5	r13
6	r14
7	r15

### 3.1.2 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 3-3](#) on page 98. The RF\_BUILD register is used to determine the configuration of core registers (see [Core Register File Configuration Register, RF\\_BUILD](#)).

**Table 3-3 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 <a href="#">Pointer Registers, GP (r26), FP (r27), SP (r28)</a> )		•
r27	FP (frame pointer) (see <a href="#">Pointer Registers, GP (r26), FP (r27), SP (r28)</a> )		•
r28	SP (stack pointer) (see <a href="#">Pointer Registers, GP (r26), FP (r27), SP (r28)</a> )		•
r29	ILINK (see <a href="#">Link Registers, ILINK (r29), BLINK (r31)</a> )		•
r30	General purpose register Note: This register is designated neither as a saved nor a temp register.		•
r31	BLINK (see <a href="#">Link Registers, ILINK (r29), BLINK (r31)</a> )		•

### 3.1.3 Multiple Register Banks



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

One can configure the number of register file banks in the CPU by using the RGF\_NUM\_BANKS option. 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].

When RGF\_NUM\_BANKS > 1, the RGF\_BANKED\_REGS option defines the number of baseline core registers that are contained in banks other than bank 0 (replicated banks).

- In RF32 configurations, the RGF\_BANKED\_REGS option can be selected from the set {4, 8, 16, 32}. In RF16 configurations, the RGF\_BANKED\_REGS option can be selected from the set {4, 8, 16}.
- The register file is always implemented as flip-flops when RGF\_NUM\_BANKS > 1.

When 16 registers are replicated they are the same as the registers in the Reduced Register Configuration set ([Reduced Configuration of Core Registers](#)) except for the ILINK register, r29.

### 3.1.3.1 Accessing Register Banks



**Note** 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, 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 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.

### 3.1.3.2 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 pre-empts 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.

### 3.1.4 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 103.

### 3.1.5 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 3-3](#) on page [98](#).

### 3.1.6 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 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 will consequently determine 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 will be the address of its delay-slot instruction plus the size of its delay-slot instruction, but always excluding any illegal LIMM operand.

### 3.1.7 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.

### 3.1.8 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 because LP\_COUNT maybe decremented if the program counter equals the loop end address. For more information about the zero delay loop mechanism, see [LPcc](#) instruction details on page [639](#).

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 3-1 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 [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.

#### **3.1.9 Reserved Register (r61)**

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

#### **3.1.10 Immediate Data Indicator, LIMM (r62)**

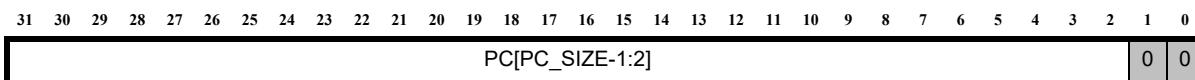
Register r62 is reserved for encoding long (32-bit) immediate (LIMM) data operands or to signify that the result of an instruction shall be discarded.. r62 is therefore not available as a general purpose register.

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 will be updated by that instruction. No actual LIMM data follows the instruction in this case.

#### **3.1.11 Word-aligned Program Counter, PCL (r63)**

**Figure 3-2 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  $32 - \text{PC\_SIZE}$  bits of PCL are always read as zero.

## 3.2 Extension Core Registers

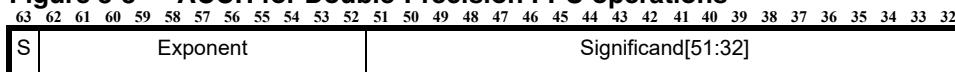
The register set is extendible in register positions 32-57 (r32-r57).

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).

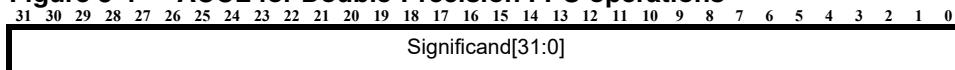
In this manual, the ACC register refers to the register pair comprising the accumulator (ACCH, ACCL), according to the definition of a register pair given in "[64-Bit Data](#)" on page [88](#).

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 3-3](#) and [3-4](#) below. For single-precision fused multiply-add or multiply-subtract operations, the ACCL register is used to represent the third operand, as shown in [Figure 3-5](#) below:

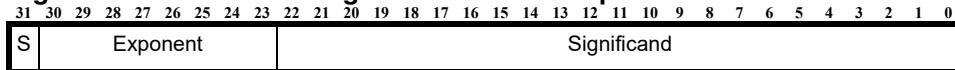
**Figure 3-3 ACCH for Double-Precision FPU operations**



**Figure 3-4 ACCL for Double-Precision FPU operations**



**Figure 3-5 ACCL for Single-Precision FPU Operations**



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

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.

### 3.2.1    **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 [99](#).

### 3.3 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. For a detailed list of the ARCv2 register set for your processor, see [Build and Auxiliary Register List](#).

#### 3.3.1 Baseline Auxiliary Registers

[Table 3-4](#) lists the baseline set of auxiliary registers. The access code definitions of these registers are explained in [Table 3-6](#).

**Table 3-4 Baseline Auxiliary Register Set**

Number	Auxiliary Register Name	Description
<b>Current Architectural State</b>		
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
<b>Saved Exception and Interrupt State</b>		
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
<b>Build Configuration Registers (BCR)</b>		
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

### 3.3.2 Optional Architectural, Exception, and Interrupt State Auxiliary Registers

[Table 3-5](#) 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 3-5 Auxiliary Register Set**

Number	Auxiliary Register Name	Description
<b>Current Architectural State</b>		
0x9	Secure Status Register, SEC_STAT	Secure status register
0x26	Secure Interrupt Vector Base Register, INT_VECTOR_BASE_S	Secure interrupt vector base address
<b>Saved Exception and Interrupt State</b>		
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
<b>Build Configuration Registers (BCR)</b>		
0x70	Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD	Build configuration for: secure interrupts

### 3.3.3 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 3-6](#). 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.



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 3-6](#) 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 3-6](#), "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 3-6 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	•	I	•	I	•	-
R	P	I	•	I	•	-
w	I	•	I	•	-	•
rw	•	•	•	•	•	•
W	I	P	I	•	-	•
rW	•	P	•	•	•	•
RW	P	P	•	•	•	•
rG	•	I	•	I	•	•
RG	P	I	•	I	•	•
NN <sup>a</sup>	P <sup>b</sup>	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 an ECR value: 0x071020.

### 3.3.4 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 LP\_SIZE parameter is non-zero, the LP\_START and, LP\_END registers are included in the auxiliary register set.

**Table 3-7 Zero Overhead Loop Auxiliary Registers (LP\_SIZE > 0)**

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

Similarly, when the CODE\_DENSITY is configured, 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 3-8 Indexed Table Auxiliary Register (CODE\_DENSITY == 1)**

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

### 3.3.5 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 3-9 XPU, User Extension Permission Auxiliary Register (APEX\_OPTION == 1)**

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 3-10 Extension Flags Register, XFLAGS**

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

### 3.3.6 Optional Build Configuration Registers

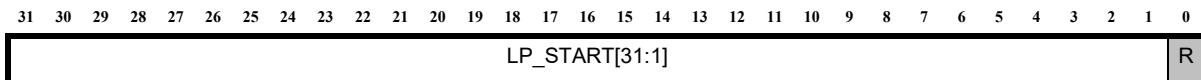
The ARCv2 architecture defines two special regions within the auxiliary address map. These are populated as required by the read-only registers which encode information about the build configuration of the system. These are located from addresses 0x60 to 0x7F and 0xC0 to 0xFF. Unused addresses in these series are all reserved.

### 3.3.7 Loop Start Register, LP\_START

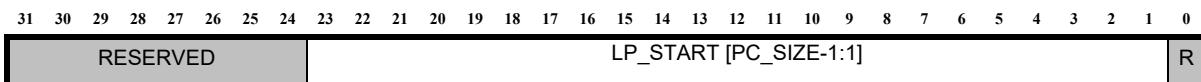
Address: 0x02

Access: rw

**Figure 3-6 LP\_START Register (PC\_SIZE == 32)**



**Figure 3-7 LP\_START Register (PC\_SIZE < 32)**



The loop start (LP\_START) register contains the address at which the current zero delay loop begins. [Figure 3-6](#) on page 109 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](#)).

LP\_START follows the auxiliary PC register (see [Program Counter, PC](#)) format, and therefore, its size is determined by the PC\_SIZE configuration parameter.

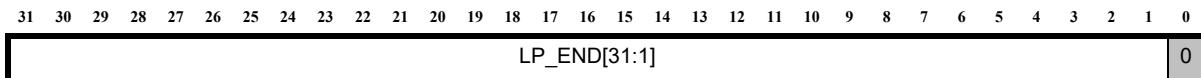
Bit 0 is reserved and must always be set to zero. Bit 0 returns zero when read by an LR instruction. Programming cautions may exist when using the loop control registers. For more information, see [LPcc](#) instruction on [page 639](#) and “[Loop Count Register, LP\\_COUNT \(r60\)](#)” on page 100.

### 3.3.8 Loop End Register, LP\_END

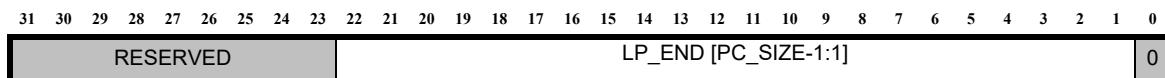
Address: 0x03

Access: rw

**Figure 3-8 LP\_END Register**



**Figure 3-9 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](#)).

LP\_END follows the auxiliary PC register (see [Program Counter, PC](#)) format, and therefore, its size is determined by the PC\_SIZE configuration parameter.

Bit 0 is reserved and must always be set to zero. Bit 0 always returns zero when read using an LR instruction. Programming cautions may exist when using the loop control registers. For more information, see the [LPcc](#) instruction on [page 639](#) and “[Loop Count Register, LP\\_COUNT \(r60\)](#)” on page [100](#).

### 3.3.9 Core Identity Register, IDENTITY

Address: 0x04

Access: r

**Figure 3-10 IDENTITY Register**

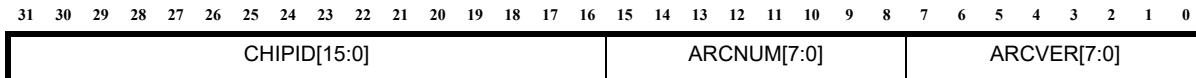


Figure 3-10 on page 111 shows the identity register (IDENTITY). This register contains the following fields:

**Table 3-11 IDENTITY Field Descriptions**

Field	Bit	Description
ARCVER	[7:0]	<b>ARC baseline instruction set version number</b> <ul style="list-style-type: none"> <li>■ 0x41: ARC EM processor version 1.1a</li> <li>■ 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.</li> <li>■ 0x43: ARC EM processor version 4.0; 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.</li> <li>■ 0x43: ARC SEM processor version 1.0; Support for the address and data bus scrambling for caches, secure kernel and user modes, side-channel protection, secure debug, micro-DMA enhancements, and so on. See the Release Notes for details.</li> <li>■ 0x50: ARC HS processor version 1.0; first release of the ARC HS processor.</li> <li>■ 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,</li> <li>■ 0x52: ARC HS version 2.1; added single-cycle option for DCCM and data cache, dropped support for BVCI,</li> <li>■ 0x53: ARC HS version 3.0 and DW 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.</li> <li>■ 0x60 to 0xFF = Reserved</li> </ul>

**Table 3-11 IDENTITY Field Descriptions (Continued)**

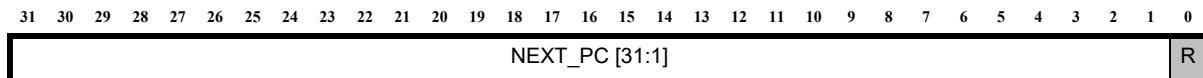
Field	Bit	Description
ARCNUM	[15:8]	<p>This field allows you to uniquely identify each core in a multi-core system.</p> <p>Values: 0 to 255</p> <p>Default: 0</p> <p>The value of this field is a reflection of the <code>-arc_num</code> option in the ARChitect tool. In a multi-core configuration, ARChitect assigns the value of the <code>-arc_num</code> option to the first core. All the other cores are assigned increasing sequential values based on their order in the chain. By default, the <code>-arc_num</code> option of the first core in the chain is assigned 0 and the other cores in the configuration are assigned core IDs 1, 2, 3, and so on.</p> <p>The input pin <code>arcnum[7:0]</code> is provided on the core interface to define the ARCNUM field.</p> <p>Additionally, 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.</p>
CHIPID	[31:16]	The unique chip identifier assigned by Synopsys.

### 3.3.10 Program Counter, PC

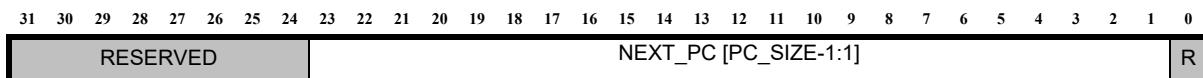
Address: 0x06

Access: rG

**Figure 3-11 PC Register Addressing Full 32-bit Address Space**



**Figure 3-12 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 3-12 lists the addressable range for different PC width configurations.

**Table 3-12 ARCv2 Addressable Range vs. PC Size**

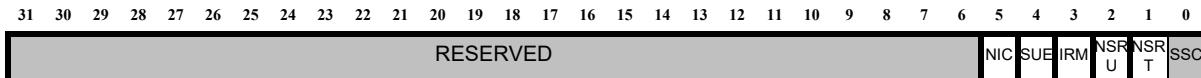
PC Width (bits)	Addressable Range
16	64 KB
20	1 MB
24	16 MB
28	256 MB
32	4 GB

### 3.3.11 Secure Status Register, SEC\_STAT

Address: 0x09

Access: See [Table 3-13](#), [Table 3-14](#), and [Table 3-14](#)

**Figure 3-13 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 3-13](#) lists the access permissions the SEC\_STAT register bits when the core is operating in the normal mode and the secure mode.



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

**Table 3-13 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 3-14](#) lists the debug access to the SEC\_STAT register when the core is operating in the secure mode.

**Table 3-14 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 3-15 SEC\_STAT Field Description**

Field	Bits	Description
SSC	[0]	<p>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:</p> <ul style="list-style-type: none"> <li>■ <a href="#">Secure Stack Region Top Address, S_STACK_TOP</a> and <a href="#">Secure Stack Region Base Address, S_STACK_BASE</a></li> </ul> <p>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.</p>
NSRT	[1]	<p>Indicates whether a function return or an interrupt return can switch a normal operating mode to a secure operating mode.</p> <ul style="list-style-type: none"> <li>■ 0x0: A function return is allowed to switch operating mode from normal to a secure operating mode.</li> <li>■ 0x1: An interrupt return can switch the operating mode from normal to a secure operating mode.</li> </ul> <p>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.</p> <p>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.</p>

**Table 3-15 SEC\_STAT Field Description (Continued)**

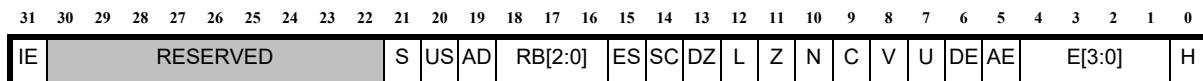
Field	Bits	Description
NSRU	[2]	<p>Indicates the operating mode that the core must return to when switching from a normal mode to the secure mode.</p> <ul style="list-style-type: none"> <li>■ 0x0: Return from the normal mode to the secure kernel mode.</li> <li>■ 0x1: Return to the normal mode to the secure user mode.</li> </ul> <p>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.</p> <p>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.</p>
IRM	[3]	<p>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.</p>
SUE	[4]	<p>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.</p>
NIC	[5]	<p>This enable bit allows the secure kernel mode code to control whether the SETI &amp; CLRI instructions are available in normal mode, and whether the SLEEP &amp; 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.</p>

### 3.3.12 Status Register, STATUS32

Address: 0x0A

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

**Figure 3-14 STATUS32 Register**



The status register, STATUS32 that 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.

**Table 3-16 STATUS32 Bit-Field Definitions and Read / Write Accessibility**

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	El_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 3-17 STATUS32 Bit-Field Definitions and Read / Write Accessibility**

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`.

## 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 86 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`.

## US

This bit controls the behavior of the WLFC, if configured via 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 WLFC 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.

## 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](#).

## 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.

## 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.

## 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.

## 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`.



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.

## 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:

- Stack Region Top Address, `STACK_TOP` and Stack Region Base Address, `STACK_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.

## 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.

## 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.

## 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.

## 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](#).

### STATUS32 Behavior on Interrupt Entry or Exit

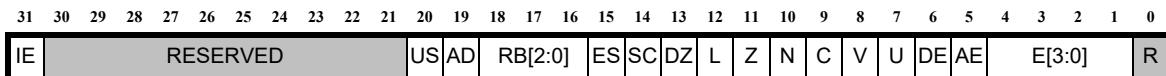
On interrupt entry, the STATUS32 register is either pushed to the stack or copied to STATUS32\_P0, depending on [FIRQ\\_OPTION](#) and the level of the interrupt (P0 or otherwise). On exception entry, the STATUS32 register is copied to ERSTATUS. However, an EV\_Trap exception clears the ES and DE fields 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.

### 3.3.13 Status Register Priority 0, STATUS32\_P0

Address: 0x0B

Access: RW

**Figure 3-15 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.

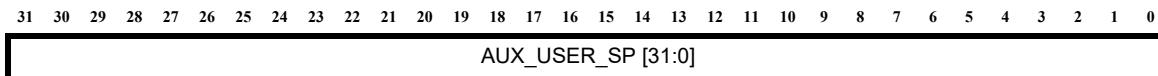
On entry to P0 interrupt, if FIRQ\_OPTION is enabled, the STATUS32 register is copied to STATUS32\_P0. On return from the highest priority interrupt, P0, the STATUS32 register is restored from STATUS32\_P0.

### 3.3.14 Saved User Stack Pointer, AUX\_USER\_SP

Address: 0x0D

Access: RW

**Figure 3-16 AUX\_USER\_SP Register**



You can read and write to the AUX\_USER\_SP register in kernel mode; this register is protected and inaccessible in user mode. This register is used to save the kernel stack pointer while in user mode. Stack pointers can be swapped as follows with a single 32-bit instruction:

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

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.

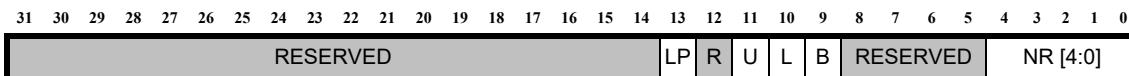
You can read and write to the AUX\_USER\_SP register in the normal and secure kernel modes; this register is protected and inaccessible in user mode. This register is used to save the normal user stack pointer when the core is switching modes from a normal user mode to the other modes. This instruction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

### 3.3.15 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

**Figure 3-17 AUX\_IRQ\_CTRL Register**



The AUX\_IRQ\_CTRL register controls the behavior of automated register save and restore or prologue and epilogue sequences during interrupt entry and exit, and context save and restore instructions.

**Table 3-18 AUX\_IRQ\_CTRL Field Description**

Field	Bit	Description
NR	[4:0]	Indicates number of general-purpose register pairs saved, from 0 to 8/16. This register saturates at 16 when the RGF_NUM_REGS == 32, and 8 when the RGF_NUM_REGS == 16. 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 <a href="#">Table 3-3</a> .
B	[9]	Indicates whether to save and restore BLINK (ignored if NR is greater than or equal to 16.)
L	[10]	Indicates whether to save and restore loop registers (LP_COUNT, LP_START, LP_END)
U	[11]	Indicates if user context is saved to user stack
LP	[13]	Indicates whether to save and restore code-density registers (EI_BASE, JLI_BASE, LDI_BASE)

You can read and write to the AUX\_IRQ\_CTRL register in kernel mode. This register is protected and inaccessible in user mode.

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 prologue or epilogue 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.

You can read and write to the AUX\_IRQ\_CTRL register in the secure kernel mode. This register is protected and inaccessible in user mode. 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 read as zero and ignored on write if only one priority level (P0) is configured. An interrupt with only one priority level does not emit a

prologue or epilogue 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.

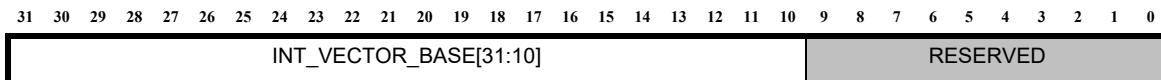
On reset, this register contains 0x00000000.

### 3.3.16 Interrupt Vector Base Register, INT\_VECTOR\_BASE

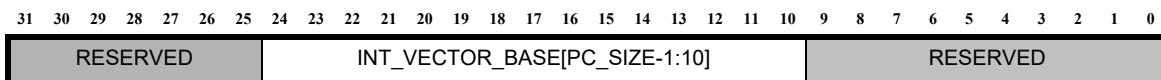
Address: 0x25

Access: RW

**Figure 3-18 INT\_VECTOR\_BASE Register (PC\_SIZE == 32)**



**Figure 3-19 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 170). 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.

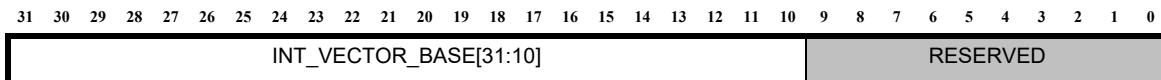
When an MMU is configured, the INT\_VECTOR\_BASE register must be set to an address in the non-translated memory space. In ARC EM family of processors, this address is within the upper 2 GBytes of the memory space. This design avoids double-fault exceptions from occurring when accessing the vector table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

### 3.3.17 Secure Interrupt Vector Base Register, INT\_VECTOR\_BASE\_S

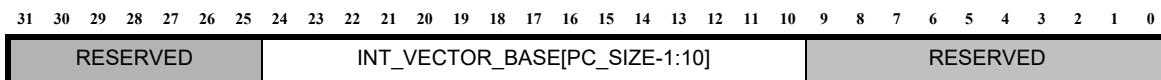
Address: 0x26

Access: RW in the secure mode

**Figure 3-20 INT\_VECTOR\_BASE\_S Register (PC\_SIZE == 32)**



**Figure 3-21 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 171). 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.

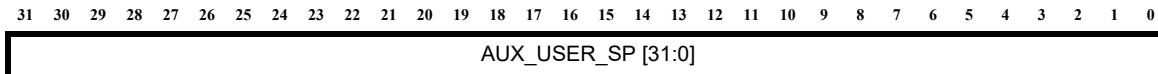
The base address for the secure interrupt table must be 1K-byte aligned. This address must be in a secure memory region as defined by the MPU. If the vector address for a secure interrupt or secure exception is in the normal memory region or the vector address for a normal interrupt or exception is in 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 a 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.

### 3.3.18 Saved Normal Kernel Stack Pointer, AUX\_KERNEL\_SP

Address: 0x38

Access: RW in the secure mode

**Figure 3-22 AUX\_KERNEL\_SP Register**



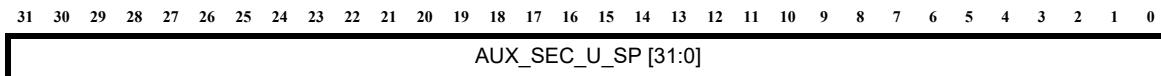
This register exists in the processor only when `-sec_modes_option==true`. You can read and write to the AUX\_KERNEL\_SP register in the normal and secure kernel modes; this register is protected and inaccessible in user mode. This register is used to save the normal kernel stack pointer when the core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

### 3.3.19 Saved Secure User Stack Pointer, AUX\_SEC\_U\_SP

Address: 0x39

Access: RW in the secure modes

**Figure 3-23 AUX\_SEC\_U\_SP Register**



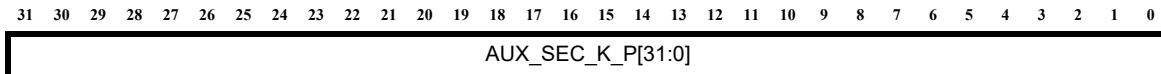
This register exists in the processor only when `-sec_modes_option==true`. You can read and write to the AUX\_SEC\_U\_SP register in the secure kernel modes; this register is protected and inaccessible in user mode. This register is used to save the secure user stack pointer when the core is switching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

### 3.3.20 Saved Secure Kernel Stack Pointer, AUX\_SEC\_K\_SP

Address: 0x3A

Access: RW in the secure modes

**Figure 3-24 AUX\_SEC\_K\_SP Register**



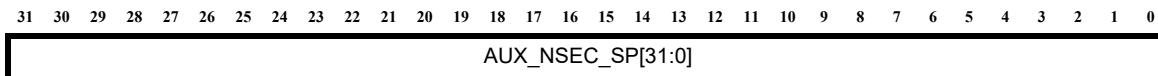
This register exists in the processor only when `-sec_modes_option==true`. You can read and write to the AUX\_SEC\_K\_SP register in the secure kernel modes; this register is protected and inaccessible in user mode. This register is used to save the secure kernel stack pointer when the core is switching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

### 3.3.21 Saved Shadow Normal Stack Pointer, AUX\_NSEC\_SP

Address: 0x3B

Access: R in the secure modes

**Figure 3-25 AUX\_NSEC\_SP Register**



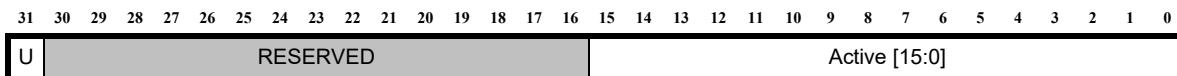
This register exists in the processor only when `-sec_modes_option==true`. This register is used to save the normal user and kernel stack pointer. When `STATUS32.U==1`, this register returns the normal user mode stack pointer, `AUX_USER_SP`. When `STATUS32.U==0`, this register returns the normal kernel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

### 3.3.22 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

**Figure 3-26 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 194.

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.

You can read and write to the AUX\_IRQ\_ACT register in kernel mode; this register is inaccessible in user mode.

When `-sec_modes_option==true`, you can read and write to the AUX\_IRQ\_ACT register in the secure kernel mode. This register is protected and inaccessible in user mode. 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 3-19 AUX\_IRQ\_ACT Field Description**

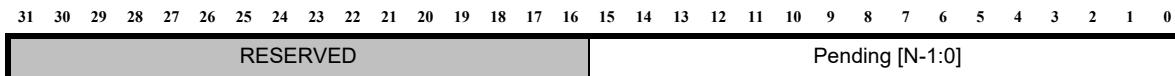
Field	Bit	Description
Active	[15:0]	Bit i indicates whether there is an active interrupt at priority i. If fewer than 16 priority levels are configured, unused bits are read as zero and ignored on write. Bits corresponding to the secure interrupts are read as zero when read from a normal kernel mode.
U	[31]	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.

### 3.3.23 Interrupt Priority Pending Register, IRQ\_PRIORITY\_PENDING

Address: 0x200

Access: R

**Figure 3-27 IRQ\_PRIORITY\_PENDING Register**



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

You can read IRQ\_PRIORITY\_PENDING register only in kernel mode. This register is protected and inaccessible in user mode.

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 3-20 IRQ\_PRIORITY\_PENDING Field Description**

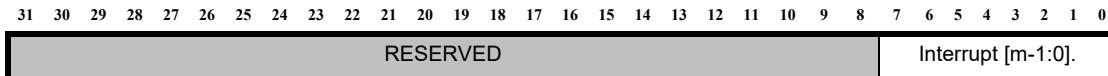
Field	Bit	Description
P	[N-1:0] Where N is the number of interrupt priority levels configured.	Bit i indicates whether there is a pending normal interrupt at priority level i. If fewer than 16 priority levels are configured, the unused bits are read as zero.

### 3.3.24 Software Interrupt Trigger, AUX\_IRQ\_HINT

Address: 0x201

Access: RW

**Figure 3-28 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 [849](#)).

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.



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.

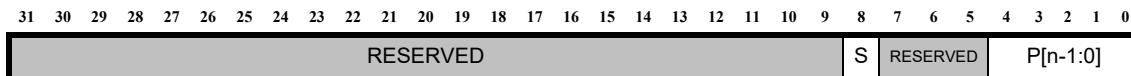
The width of the interrupt fields is determined by the number of interrupts configured in the system. If M is the number of interrupts configured,  $m=\text{ceil}(\log_2(M + 16))$ , where ceil is the function to round to the next higher integer.

### 3.3.25 Interrupt Priority Register, IRQ\_PRIORITY

Address: 0x206

Access: When SecureShield 2+2 mode is not configured: RW  
When SecureShield 2+2 mode is configured: RW in the secure mode only

**Figure 3-29 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. If an interrupt priority > N-1 is assigned, the associated interrupt priority is 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 3-21 IRQ\_PRIORITY Field Description**

Field	Bit	Description
P	[n-1:0] If N is the number of interrupt priority levels configured, $n=\text{ceil}(\log_2(N))$	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.

**Table 3-21 IRQ\_PRIORITY Field Description**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
S	[8]	<p>Indicates if the interrupt is a secure interrupt or normal interrupt. This bit is available only if <code>-sec_mode_options==true</code>.</p> <ul style="list-style-type: none"> <li>■ 0x0: generates a normal interrupt.</li> <li>■ 0x1: generates a secure interrupt.</li> </ul> <p>The timer 0 and timer 1 interrupts are by default set as normal interrupts (<math>S==0x0</math>). Other interrupts are by default assumed to be secure interrupts (<math>S==1</math>).</p> <p>This bit is read as zero and ignored on writes in the normal mode.</p>

### 3.3.26 Secure Jump and Link Indexed Top Address, NSC\_TABLE\_TOP

Address: 0x268

Access: RW in the secure mode

**Figure 3-30 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.

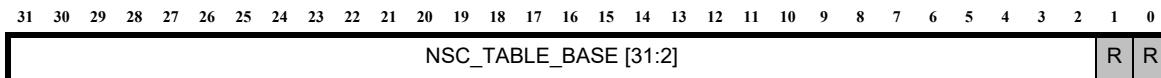
The width of this register is determined by the PC size. If pc\_size is 16 or 24, the unused MSB bits of NSC\_TABLE\_BASE/NSC\_TABLE\_TOP are read as zero and ignored on writes.

### 3.3.27 Secure Jump and Link Indexed Base Address, NSC\_TABLE\_BASE

Address: 0x269

Access: RW in the secure mode

**Figure 3-31 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.

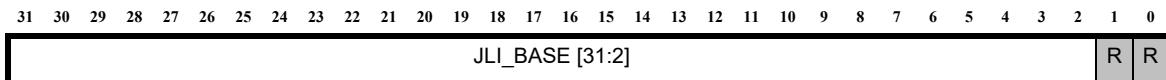
The width of this register is determined by the PC size. If `pc_size` is 16 or 24, the unused MSB bits of NSC\_TABLE\_BASE/NSC\_TABLE\_TOP are read as zero and ignored on writes.

### 3.3.28 Jump and Link Indexed Base Address, JLI\_BASE

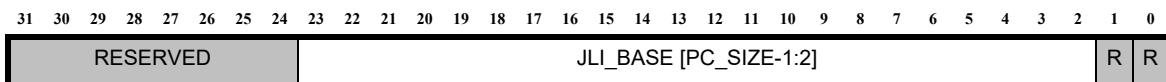
Address: 0x290

Access: rw

**Figure 3-32 JLI\_BASE Register when PC\_SIZE == 32**



**Figure 3-33 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.

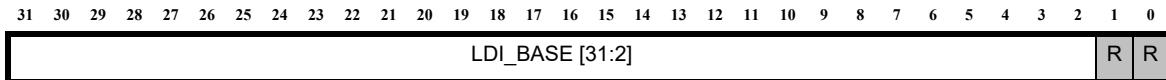
The width of the JLI\_BASE register is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

### 3.3.29 Load Indexed Base Address, LDI\_BASE

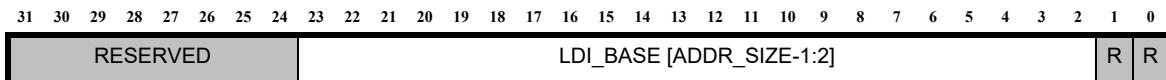
Address: 0x291

Access: rw

**Figure 3-34 LDI\_BASE Register when ADDR\_SIZE == 32**



**Figure 3-35 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.

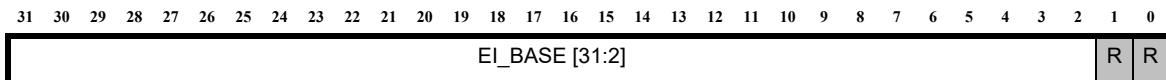
The width of the LDI\_BASE register is determined by the address size. Reading any reserved bits returns 0 and writes to such bits have no effect.

### 3.3.30 Execute Indexed Base Address, EI\_BASE

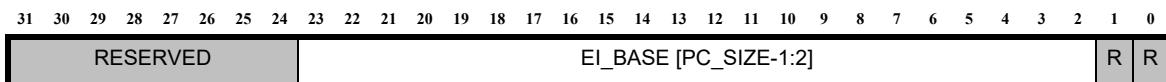
Address: 0x292

Access: rw

**Figure 3-36 EI\_BASE Register when PC\_SIZE == 32**



**Figure 3-37 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.

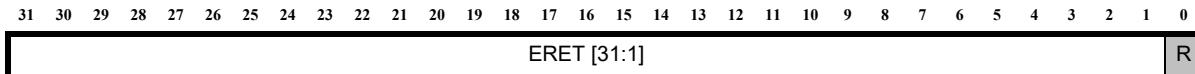
The width of the EI\_BASE register is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

### 3.3.31 Exception Return Address, ERET

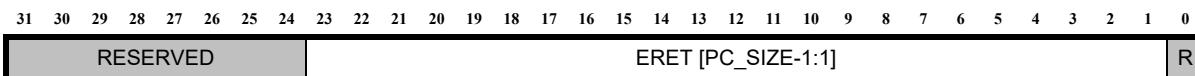
Address: 0x400

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

**Figure 3-38 ERET Register when PC\_SIZE == 32**



**Figure 3-39 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 or interrupt is taken. This value is always the address of the next instruction to be executed in program order. Therefore, when an exception is raised due to a faulty instruction, the faulty instruction does not complete. Therefore, ERET is set to the address of the faulty instruction. However, 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.

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.

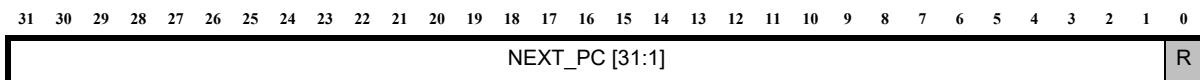
When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV\_PrivilegeV violation exception. When an exception is triggered by an instruction in the normal mode,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

### 3.3.32 Exception Return Branch Target Address, ERBTA

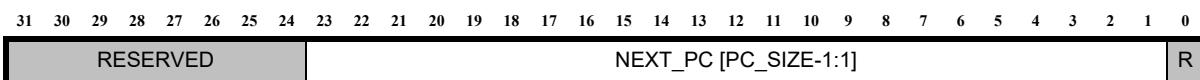
Address: 0x401

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

**Figure 3-40 ERBTA Register when PC\_SIZE == 32**



**Figure 3-41 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.

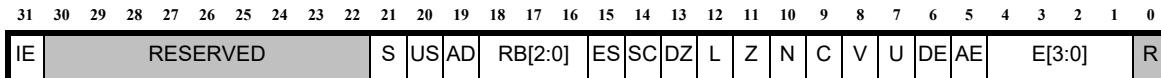
When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV\_PrivilegeV violation exception. When an exception is triggered by an instruction in the normal mode,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

### 3.3.33 Exception Return Status, ERSTATUS

Address: 0x402

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

**Figure 3-42 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.

### 3.3.34 Exception Cause Register, ECR

Address: 0x403

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

**Figure 3-43 ECR 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
P	U	RESERVED		Vector Number				Cause Code				Parameter																			

The Exception Cause register (ECR) provides information about the source of the most recent exception to have been taken. [Figure 3-43](#) on page 147 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 4-6](#) on page 222 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 C, “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 C, “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 prologue.

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



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

Please refer to either [Appendix C.1.1, “ECR User-mode Setting in the ARC EM Family of Cores”](#) or [Appendix C.3.1, “ECR User-mode Setting for the ARC EM 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.

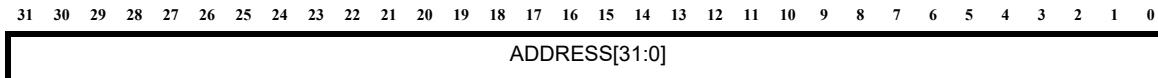
When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV\_PrivilegeV violation exception. When an exception is triggered by an instruction in the normal mode,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

### 3.3.35 Exception Fault Address, EFA

Address: 0x404

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

**Figure 3-44 EFA Register**



**Note** Configurations of the ARC EM family processors without a memory protection unit (MPU), or other forms of memory protection or address translation, do not support the EFA register.

When MMU is configured and a memory access triggers an exception, the exception fault address register (EFA) is loaded with the address that triggered the exception. For example, when a memory access spans an MMU page boundary, if the next page access causes the exception, the EFA is loaded with the base address of that next page. For non-MMU exceptions, the EFA register is loaded with the PC value used to fetch the faulting instruction or the first address. Exceptions triggered by watchpoints are imprecise and the EFA register is thus undefined on their occurrence. 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.

The EFA register sometimes holds a physical address and sometimes a virtual address. There are a number of memory-related exceptions that have an associated fault address. These exceptions normally update EFA, and normally update it with the virtual address. However, there are two exceptions to this rule:

- A bus error exception caused by an uncached load or store operation sets EFA to the physical address of the faulting memory location. EFA is always virtual address for the instruction fetch related memory exceptions.
- When an internal instruction memory error exception occurs, the EFA contains the address of the instruction that contains the unrecoverable error. In case of an internal data memory error exception the EFA register contains the address of the data item address for which the unrecoverable error occurred.
- An exception occurring due to a data cache copy-back operation, which may be due to either a bus error or an irrecoverable ECC or parity error within the data cache, updates EFA with a physical address as follows:
  - If the exception was due to an ECC or parity error, detected when reading the tag of the cache line to be copied back, then the value assigned to EFA will include the faulty bit(s), and therefore the tag field within the EFA value cannot be trusted as the copy-back operation will not have taken place. If an ECC or parity error is detected when reading data from the data cache while copying a line back to memory, the value assigned to EFA will be correct.

- If the exception was due to a bus error reported during a copy-back bus write transaction, then EFA indicates the physical address of the cache line that was being copied back.

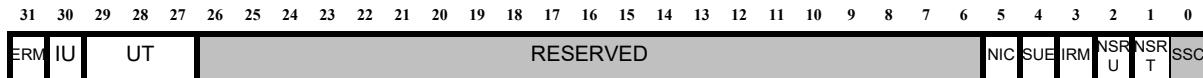
When an exception is triggered by an instruction in the secure mode, an internal bit is set to 1 on exception entry. In such cases, this register cannot be accessed in the normal mode, and raises an EV\_Privilege violation exception. When an exception is triggered by an instruction in the normal mode,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

### 3.3.36 Exception Secure Status Register, ERSEC\_STAT

Address: 0x406

Access: Bit-specific; see the bit description

**Figure 3-45 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 3-22 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 3-23 ERSEC\_STAT Field Description**

<b>Field</b>	<b>Bits</b>	<b>Description</b>
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: <ul style="list-style-type: none"> <li>■ 0x0: secure to normal call</li> <li>■ 0x1: normal to secure return</li> <li>■ 0x2: normal to secure call</li> <li>■ 0x3: secure to normal return</li> <li>■ 0x4: exception prologue</li> <li>■ 0x5: exception epilogue</li> <li>■ 0x6: IRQ prologue</li> <li>■ 0x7: IRQ epilogue</li> </ul>
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.

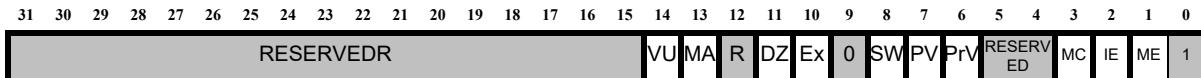
### 3.3.37 Secure Exception Register, AUX\_SEC\_EXCEPT

Address: 0x407

Access: RW in the secure mode

Default 0x00000000

**Figure 3-46 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 3-24 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]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only.</li> </ul>
Instruction Error (IE)	[2]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only</li> </ul>
EV_MachineCheck (MC)	[3]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only</li> </ul>
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 3-24 AUX\_SEC\_EXCEPT Field Description (Continued)**

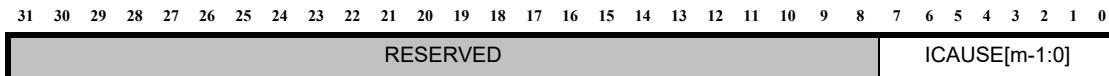
<b>Field</b>	<b>Bits</b>	<b>Description</b>
EV_ProtV (PrV)	[6]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only</li> </ul>
EV_PrivilegeV (PV)	[7]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only</li> </ul>
EV_SWI (SW)	[8]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only</li> </ul>
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]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only</li> </ul>
EV_DivZero (DZ)	[11]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only</li> </ul>
EV_DCError	[12]	This bit is reserved. It is always read as zero and ignored on writes.
EV_Misaligned (MA)	[13]	<ul style="list-style-type: none"> <li>■ 0x0: This exception vector is handled in the mode that triggered the exception.</li> <li>■ 0x1: This exception vector is handled in the secure mode only</li> </ul>
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.

### 3.3.38 Interrupt Cause Registers, ICAUSE

Address: 0x40A

Access: R

**Figure 3-47 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 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.

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.

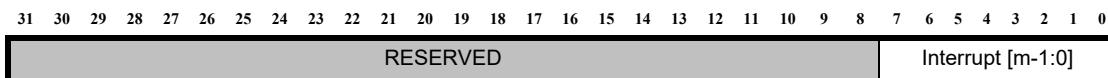
On reset, this register contains 0x00000001 for configured interrupts; if Interrupt [m-1:0] is greater than  $M+16-1$  or less than 16, this register is read as zero. Where, M is the number of interrupts configured,  $m=\text{ceil}(\log_2(M + 16))$ , and ceil is the function to round to the next higher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

### 3.3.39 Interrupt Select, IRQ\_SELECT

Address: 0x40B

Access: RW

**Figure 3-48 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 3-25 IRQ\_SELECT Field Description**

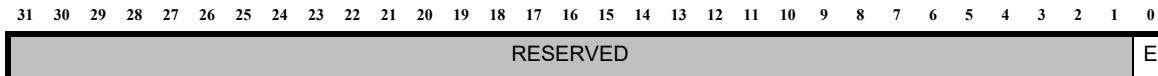
Field	Bit	Description
Interrupt	[m-1:0]	Selects a specific interrupt. If M is the number of interrupts configured, $m=\text{ceil}(\log_2(M + 16))$

### 3.3.40 Interrupt Enable Register, IRQ\_ENABLE

Address: 0x40C

Access: RW

**Figure 3-49 IRQ\_ENABLE Register**



**Table 3-26 IRQ\_ENABLE Field Description**

Field	Bit	Description
E	[0]	Value of IRQ_ENABLE for the selected interrupt.

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

You can read and write to IRQ\_ENABLE in kernel mode; this register is protected and inaccessible in user mode.

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.

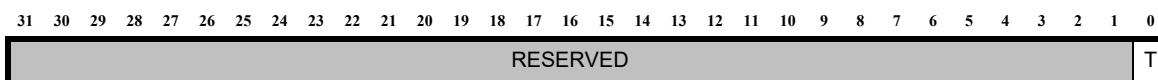
On reset, this register contains 0x00000001 for configured interrupts. Therefore, all configured interrupts are individually enabled upon reset. If Interrupt [m-1:0] is greater than M+16-1 or less than 16, this register is read as zero and ignored on write. Where, M is the number of interrupts configured,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

### 3.3.41 Interrupt Trigger Register, IRQ\_TRIGGER

Address: 0x40D

- Access:
- When SecureShield 2+2 mode is not configured: RW
  - When SecureShield 2+2 mode is configured:
    - RW in the normal and secure modes for normal interrupts
    - RW in the secure mode for the secure interrupts

**Figure 3-50 IRQ\_TRIGGER Register**



**Table 3-27 IRQ\_TRIGGER Field Description**

Field	Bit	Description
T	[0]	Value of IRQ_TRIGGER, for the selected interrupt

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.

You can read and write to IRQ\_TRIGGER in kernel mode; this register is protected and inaccessible in user mode.

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.

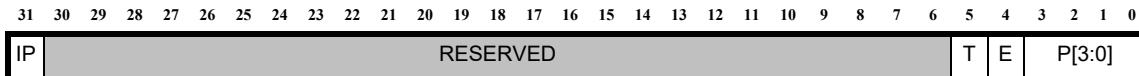
On reset, this register contains 0x00000000 for configured interrupts; if Interrupt [m-1:0] is greater than M+16-1 or less than 16, this register is read as zero and ignored on write. Where, M is the number of interrupts configured, m=ceil(log<sub>2</sub>(M + 16)), and ceil is the function to round to the next higher integer.

### 3.3.42 Interrupt Status Register, IRQ\_STATUS

Address: 0x40F

Access: R

**Figure 3-51 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 3-28 IRQ\_STATUS Field Description**

Field	Bit	Description
P	[3:0]	Value of <a href="#">Interrupt Priority Register, IRQ_PRIORITY</a> , for the selected interrupt
E	[4]	Value of <a href="#">Interrupt Enable Register, IRQ_ENABLE</a> , for the selected interrupt
T	[5]	Value of <a href="#">Interrupt Trigger Register, IRQ_TRIGGER</a> , for the selected interrupt
IP	[31]	Value of <a href="#">Interrupt Pending Register, IRQ_PENDING</a> , for the selected interrupt. Bit 31 allows quick check for pending interrupt using the .MI condition or less-than-zero test.

### 3.3.43 User Mode Extension Enable Register, XPU

Address: 0x410

Access: RW

**Figure 3-52 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	u14	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 Privilege Violation 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

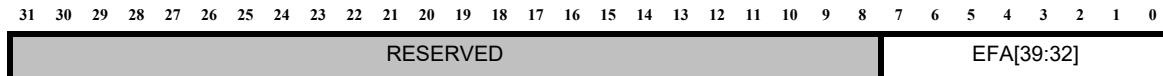
On Reset, the XPU register is set to 0x00000000, so that all extension groups are initially protected from User mode access. For more information about the allocation of XPU bits to each extension group, see the processor-specific Databook.

### 3.3.44 Exception Fault Address, EFA\_EXT

Address: 0x411

Access: RW

**Figure 3-53 EFA\_EXT Register**



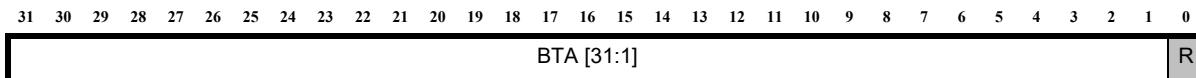
This register exists only if the physical address extension is enabled. This register stores the most significant 8 bits of the 40-bit physical address from the exception fault address, and the lower 32-bits are stored in the [EFA](#).

### 3.3.45 Branch Target Address, BTA

Address: 0x412

Access: RW

**Figure 3-54 BTA Register (PC\_SIZE == 32)**



**Figure 3-55 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

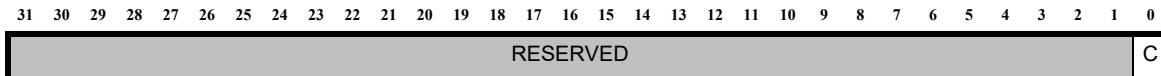
The Execute Indexed instruction, EI\_S, causes the next instruction to be fetched from a table of instructions, located in memory at an address given by the EI\_BASE auxiliary register. The instruction fetched from the table is said to be in the execution slot of the EI\_S instruction. Not all instructions 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.

### 3.3.46 Interrupt Pulse Cancel Register, IRQ\_PULSE\_CANCEL

Address: 0x415

Access: W

**Figure 3-56 IRQ\_PULSE\_CANCEL Register**



**Table 3-29 IRQ\_PULSE\_CANCEL Field Description**

Field	Bit	Description
C	[0]	Value to be written to IRQ_PULSE_CANCEL for the selected interrupt.

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-cancelling value 0 is always ignored. A write of pulse-cancelling 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.

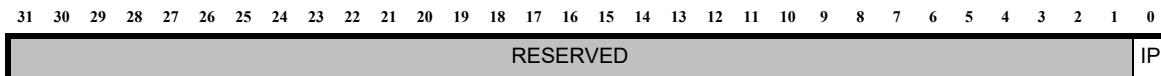
For secure interrupts, the corresponding IRQ\_PULSE\_CANCEL register can be written from the secure kernel mode only. For secure interrupts, attempts to write the corresponding IRQ\_PULSE\_CANCEL register from the normal kernel mode are ignored. However, in the secure kernel mode, you can write to the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

### 3.3.47 Interrupt Pending Register, IRQ\_PENDING

Address: 0x416

Access: R

**Figure 3-57 IRQ\_PENDING Register**



**Table 3-30 IRQ\_PENDING Field Description**

Field	Bit	Description
IP	[0]	Value of IRQ_PENDING for the selected interrupt

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.

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.

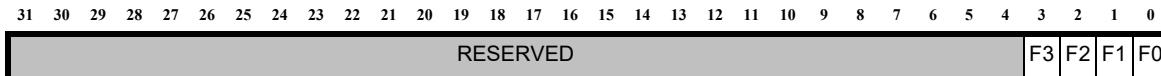
On reset, this register contains 0x00000000 for configured interrupts; if Interrupt [m-1:0] is greater than M+16-1 or less than 16, this register is read as zero. Where, M is the number of interrupts configured, m=ceil(log<sub>2</sub>(M + 16)), and ceil is the function to round to the next higher integer.

### 3.3.48 User Extension Flags Register, XFLAGS

Address: 0x44F

Access: RW

**Figure 3-58 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.

## 3.4 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 Only Access](#)).

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

[Table 3-31](#) summarizes the build configuration registers for components that are described in this document.

**Table 3-31 Build Configuration Registers**

<b>Number</b>	<b>Name</b>	<b>Access</b>	<b>Description</b>
0x60	Build Configuration Registers Version, BCR_VER	R	Build Configuration Registers Version
0x63	BTA Configuration Register, BTA_LINK_BUILD	R	Build configuration for BTA LINK
0x68	Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD	R	Build configuration for: Interrupt Vector Base
0x70	Secure Interrupt Vector Base Address Configuration, SEC_VECBASE_BUILD	SR	Build configuration for the secure interrupt vector base
0x6E	Core Register File Configuration Register, RF_BUILD	R	Build configuration for: Core Register File
0x7B	Multiplier Configuration Register, MULTIPLY_BUILD	R	Build configuration for: Multiply
0x7C	Swap Instruction Configuration Register, SWAP_BUILD	R	Build configuration for: Swap
0x7D	Normalize Instruction Configuration Register, NORM_BUILD	R	Build configuration for: Normalize
0x7E	Min/Max Instruction Configuration Register, MINMAX_BUILD	R	Build configuration for: MIN/MAX
0x7F	Barrel Shifter Configuration Register, BARREL_BUILD	R	Build configuration for: Barrel Shift
0xC1	Instruction Set Configuration Register, ISA_CONFIG	R	Build configuration for: ISA configuration
0xF3	Interrupt Build Configuration Register, IRQ_BUILD	R	Build configuration for: Interrupts

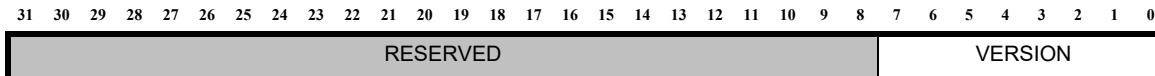
### 3.4.1 Build Configuration Registers Version, BCR\_VER

Address: 0x60

Access: R

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

**Figure 3-59 BCR\_VER Register**



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 3-32 BCR\_VER Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of Build Configuration Registers</b> 0x01 = Indicates that the BCR Region is at addresses 0x60-0x7F and ISA_CONFIG only 0x02 = Indicates that the BCR Region is at addresses 0x60-0x7F and 0xC0-0xFF The other values for this field are reserved.

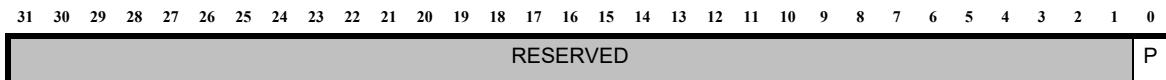
### 3.4.2 BTA Configuration Register, BTA\_LINK\_BUILD

Address: 0x63

Access: R

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

**Figure 3-60 BTA\_LINK\_BUILD Register**



**Table 3-33 BTA\_LINK\_BUILD Field Descriptions**

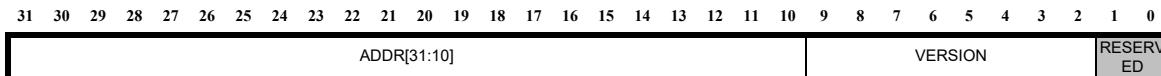
Field	Bit	Description
P	[0]	<b>Presence of BTA Registers</b> 0x0 = BTA_L1 and BTA_L2 registers are absent All other values are reserved.

### 3.4.3 Interrupt Vector Base Address Configuration, VECBASE\_AC\_BUILD

Address: 0x68

Access: R

**Figure 3-61 VECBASE\_AC\_BUILD Register**



**Table 3-34 VECBASE\_AC\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[9:2]	<b>Version of Interrupt Unit</b> 0x04 = ARCv2 Interrupt Unit
ADDR	[31:10]	Interrupt Vector Base Address. This value of the ADDR field is configured at build time through the -intvbase_preset parameter.

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.

VECBASE\_AC\_BUILD is a read-only register. This value of the ADDR field is configured at build time through the -intvbase\_preset parameter.

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

The reset vector, the first entry in the vector table, is located at the 1K-byte aligned address given in the ADDR field of this register. The processor begins execution after reset by fetching the instruction at the address referenced by the reset vector.

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 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.

### 3.4.4 Secure Interrupt Vector Base Address Configuration, SEC\_VECBASE\_BUILD

Address: 0x70

Access: R in the secure mode

**Figure 3-62 SEC\_VECBASE\_BUILD Register**



**Table 3-35 SEC\_VECBASE\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[9:2]	<b>Version of Interrupt Unit with secure interrupt table</b> 0x01 = ARCV2 Interrupt Unit
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.

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.

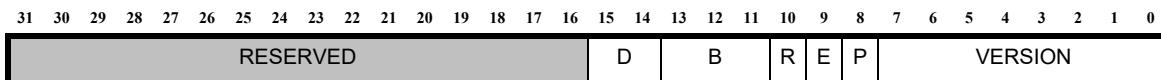
### 3.4.5 Core Register File Configuration Register, RF\_BUILD

Address: 0x6E

Access: R

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.

**Figure 3-63 RF\_BUILD Register**



**Table 3-36 RF\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of Core Register Set</b> 0x02 = Current Version
P	[8]	<b>Number of Ports</b> 0x0 = 3-port register file 0x1 = 4-port register file
E	[9]	<b>Number of Entries</b> 0x0 = 32-entry register file 0x1 = 16-entry register file
R	[10]	<b>Reset State</b> 0x0 = Not cleared on reset 0x1 = Cleared on reset
B	[13:11]	<b>Number of Register Banks in addition to the main core register bank</b>  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 = 6 additional register bank 0x7 = 7 additional register banks

**Table 3-36 RF\_BUILD Field Descriptions (Continued)**

Field	Bit	Description
D	[15:14]	<b>Number of duplicated registers in each additional register bank</b> 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.

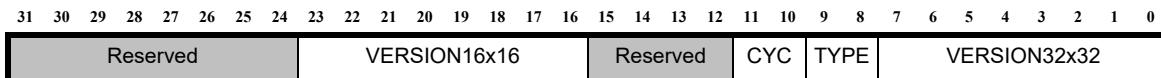
### 3.4.6 Multiplier Configuration Register, MULTIPLY\_BUILD

Address: 0x7B

Access: R

The Multiplier configuration register (MULTIPLY\_BUILD) contains the version of the multiply instructions.

**Figure 3-64 MULTIPLY\_BUILD Register**



**Table 3-37 MULTIPLY\_BUILD Field Descriptions**

Field	Bit	Description
VERSION32x32	[7:0]	<b>Version of 32x32 Multiply</b> 0x06 = ARCv2 32x32 Multiply
TYPE[1:0]	[9:8]	<b>Type of Multiply</b> 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]	<b>Number of Cycles</b> 0x0 = 1 cycle 0x1 = 2 cycles 0x2 = 3 cycles 0x3 = 4 cycles
DSP [3:0]	[15:12]	DSP capabilities 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 All other values are reserved
VERSION16x16	[23:16]	<b>Version of 16x16 Multiply</b> 0x02 = ARCv2 16x16 Multiply

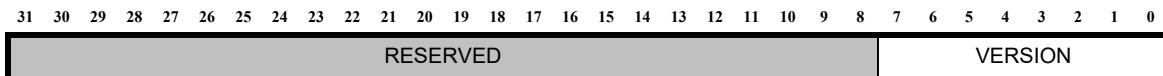
### 3.4.7 Swap Instruction Configuration Register, SWAP\_BUILD

Address: 0x7C

Access: R

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

**Figure 3-65 SWAP\_BUILD Register**



**Table 3-38 SWAP\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of Swap</b> 0x01 = ARCompact V1 version of SWAP option 0x03 = ARCv2 version of SWAP option

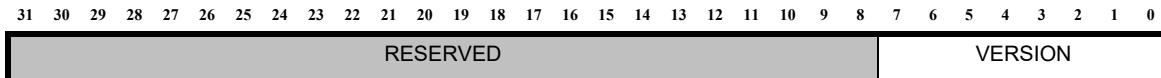
### 3.4.8 Normalize Instruction Configuration Register, NORM\_BUILD

Address: 0x7D

Access: R

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

**Figure 3-66 NORM\_BUILD Register**



**Table 3-39 NORM\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of Norm</b> 0x01 = Reserved 0x02 = ARCompact V1 version 0x03 = ARCv2 version

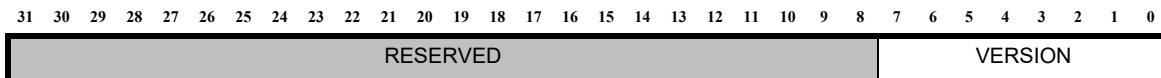
### 3.4.9 Min/Max Instruction Configuration Register, MINMAX\_BUILD

Address: 0x7E

Access: R

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

**Figure 3-67 MINMAX\_BUILD Register**



**Table 3-40 MINMAX\_BUILD field descriptions**

Field	Bit	Description
VERSION	[7:0]	Version of Min/Max 0x01 = Reserved 0x02 = Current Version

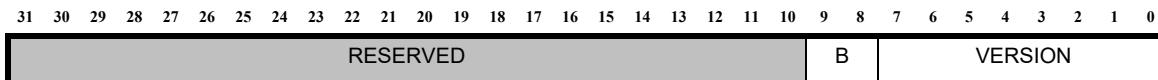
### 3.4.10 Barrel Shifter Configuration Register, BARREL\_BUILD

Address: 0x7F

Access: R

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

**Figure 3-68 BARREL\_BUILD Register**



**Table 3-41 BARREL\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of Barrel Shifter</b> 0x01 = Reserved 0x02 = ARCompact V1 version 0x03 = ARCv2 version
B[1:0]	[9:8]	<b>SHIFT_OPTION</b> 0 = Single-bit shifts only 1 = Single-bit and shift-assist instructions 2 = Single-bit and barrel-shift instructions 3 = All shift instructions

### 3.4.11 Instruction Set Configuration Register, ISA\_CONFIG

Address: 0xC1

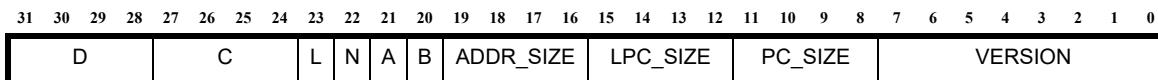
Access: R

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 67.

In addition to the configured build options, the Version field indicates the version number for the ISA\_CONFIG register itself. 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 67, and Table 1-2.

**Figure 3-69 ISA\_CONFIG Build Register**



**Table 3-42 ISA\_CONFIG Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	0x01 =ARCompact V1 version 0x02 =ARCv2 version
PC_SIZE	[11:8]	0000 =16-bit width 0001 =20-bit width 0010 =24-bit width 0011 =28-bit width 0100 =32-bit width
LPC_SIZE	[15:12]	0000 =Zero overhead loop not supported 0001 =8-bit width 0010 =12-bit width 0011 =16-bit width 0100 =20-bit width 0101 =24-bit width 0110 =28-bit width 0111 =32-bit width

**Table 3-42 ISA\_CONFIG Field Descriptions (Continued)**

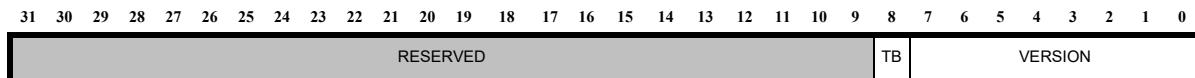
<b>Field</b>	<b>Bit</b>	<b>Description</b>
ADDR_SIZE	[19:16]	0000 =16-bit width 0001 =20-bit width 0010 =24-bit width 0011 =28-bit width 0100 =32-bit width
B (BYTE_ORDER)	[20]	0 = Little-Endian byte ordering 1 = Big-Endian byte ordering
A (ATOMIC_OPTION)	[21]	0 = LLOCK and SCOND instructions are absent 1 = LLOCK and SCOND instructions are present
N (NON_ALIGNED)	[22]	0 = All data memory references must use natural alignment 1 = Non-aligned memory references are supported
L (LL64_OPTION)	[23]	0 = LDD and STD extension instructions are not included in the build 1 = LDD and STD instructions are included in the build
C (CODE_DENSITY_OPTION)	[27:24]	0 = Code density optional instructions are absent 2 = Code density instructions version 2 are present = All other values are reserved
D (DIV_Rem_OPTION)	[31:28]	0 = DIV/REM instructions are absent 1 = Bit-serial DIV, DIVU, REM and REMU implementation 2 = Radix-4 fast DIV, DIVU, REM and REMU implementation All other values are reserved

### 3.4.12 Core Architecture Build Configuration Register, ARCH\_CONFIG

Address: 0xF1

Access: R

**Figure 3-70 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 3-43 ARCH\_CONFIG Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version</b> <ul style="list-style-type: none"> <li>■ 0x01: First version</li> </ul>
TB	[8]	<b>Specifies the</b> build-time configuration of the turbo boost option. <ul style="list-style-type: none"> <li>■ 0x0: Turbo boost is not configured.</li> <li>■ 0x1: Turbo boost is configured.</li> </ul>

### 3.4.13 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, and how many of those interrupts have external interrupt pins.

**Figure 3-71 IRQ\_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	F	P[3:0]	EXTS[7:0]							IRQS[7:0]							VERSION														

**Table 3-44 IRQ\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of the Interrupt Controller</b> 0x01 = ARC v2
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

IRQ\_BUILD is read-only in kernel mode, with normal BCR access protections.



The reset values 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.

This register is present in a build only if the processor is configured to include interrupts.

# 4

# Interrupts and Exceptions

## 4.1 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. The secure interrupts and exceptions cause the processor to enter the secure kernel operating mode, and normal interrupts and exception cause the processor to enter the normal kernel operating mode. When the secure modes option is disabled, the processor, upon returning from an interrupt or an exception, enters 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.

## 4.2 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.

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.

#### 4.2.1 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.

#### 4.2.2 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.

#### 4.2.3 Privilege Violations

This section describes the privileges available to the tasks running in user mode and kernel mode. [Table 4-1](#) 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 4-1 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 4-1 Overview of Privileges in Kernel and User Modes (Continued)**

<b>Function</b>	<b>User Mode</b>	<b>Kernel Mode</b>
Access to Basecase Auxiliary Registers	Only LP_START, LP_END, PC32 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

#### 4.2.3.1 Privileged Instructions

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

- [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 4-2](#) describes the privilege access of instructions in the secure mode.

**Table 4-2 Privilege Access of Instructions in the Secure Mode**

<b>Instruction</b>	<b>Access Privilege</b>
SLEEP	<ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> <li>■ Using the SLEEP instruction in the normal or secure user modes, raises an EV_PrivilegeV exception.</li> </ul>

**Table 4-2 Privilege Access of Instructions in the Secure Mode**

<b>Instruction</b>	<b>Access Privilege</b>
RTIE	Available only in the kernel modes.
CLRI	<ul style="list-style-type: none"> <li>■ When SEC_STAT.NIC==0:           <ul style="list-style-type: none"> <li>- The CLRI instruction is allowed only in the secure kernel mode.</li> <li>- Using the CLRI instruction in any other mode raises a Privilege Violation exception (0x071000).</li> </ul> </li> <li>■ When SEC_STAT.NIC==1:           <ul style="list-style-type: none"> <li>- The CLRI instruction is available in the secure kernel and normal kernel modes.</li> <li>- Using the CLRI instruction in the secure or normal user mode raises a Privilege Violation exception (0x071020).</li> </ul> </li> </ul>
SETI	<ul style="list-style-type: none"> <li>■ When SEC_STAT.NIC==0:           <ul style="list-style-type: none"> <li>- The SETI instruction is allowed only in the secure kernel mode</li> <li>- Using the SETI instruction in any other mode raises a Privilege Violation exception (0x071000).</li> </ul> </li> <li>■ When SEC_STAT.NIC==1:           <ul style="list-style-type: none"> <li>- The SETI instruction is allowed in the secure kernel and normal kernel modes.</li> <li>- Using the SETI instruction in the secure or normal user mode raises a Privilege Violation exception (0x071020).</li> </ul> </li> </ul>
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.
WEVT	<ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> <li>■ Using the WEVT instruction in the normal or secure user modes, raises an EV_PrivilegeV exception.</li> </ul>

### 4.2.3.2 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 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.

### 4.2.4 Switching Between Operating Modes and Privilege Levels

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

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.

#### 4.2.4.1 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; when a normal interrupt is triggered, the core switches from the normal user mode to the normal kernel mode. When a secure interrupt is triggered, the core switches from the secure user mode to the secure kernel mode.
- 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 the machine status register indicates that the highest priority active interrupt was taken from user mode.

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).

## 4.2.5 Switching Between Operating Modes

### 4.2.5.1 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.

### 4.2.5.2 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.

#### 4.2.5.3 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.

#### 4.2.5.4 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.

#### 4.2.6 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.

## 4.3 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:

- Timer – triggered by one of the optional extension timers and watchdog timer
- External – available as input pins to the core
- Software-only – triggered by software only

This section explains the following topics:

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

### 4.3.1 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
- Support for up to 16 interrupt priority levels
  - 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
- Interrupt handlers 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

### 4.3.2 Interrupt Unit Configuration

You can configure the interrupt controller using the options in [Table 4-3](#):

**Table 4-3 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 number of interrupts in the interrupt controller.
NUMBER_OF_LEVELS	16 (0 to 15 interrupt priorities)	Defines the number of interrupt priority levels in the interrupt controller.
EXTERNAL_INTERRUPTS	0 to 240	Defines the number of external interrupts.
FIRQ_OPTION	0, 1	Indicates whether the fast interrupt option is enabled. 0—indicates that the fast interrupt option is disabled. 1—indicates that the fast interrupt option is enabled.



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

#### 4.3.2.1 Interrupt Configuration Constraints

The number of interrupts configured must be greater than the following configurations:

- Number of timers configured in the processor; interrupts from TIMER0 and TIMER1, when present, have fixed assignments to the lowest numbered vectors - vector 16 for TIMER0 and 17 for TIMER1.
- One interrupt for the watchdog timer, if configured. The watchdog timer has a fixed assignment to vector 18.
- Number of secure interrupts configured in the processor. When secure timers are configured, secure timer 0, and secure timer 1, have fixed assignments: 20 for secure timer 0, and 21 for secure timer 1.
- 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. When secure timers are not configured, the internal interrupts are assigned vector starting 20. When secure timers are configured, the internal interrupts are assigned at vector starting 22.
  - 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  $(2 * \text{DMAC\_CHANNELS})$ . When secure timers are not configured, the internal interrupts are assigned vector starting 20 if they are unused and only if all the DMA interrupts can be assigned as a block.

If the interrupts cannot be assigned as a block starting vector 20, the lowest available block is chosen. When secure timers are configured, the internal interrupts are assigned at vector starting 22.

- Multiple-External: Two signals for each DMA channel: one for error and one for completion. The number of interrupts assigned is  $(2 * \text{DMAC\_CHANNELS})$ . The external interrupts are assigned as a block to the lowest available unused interrupt vectors.
- Number of configured external interrupts. External interrupts are assigned any non-timer interrupt number, 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, 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.

#### 4.3.2.2 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 endianess.

[Table 4-6](#) 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.

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.

#### 4.3.2.3 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](#).

### 4.3.3 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.

#### 4.3.3.1 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.

#### 4.3.3.2 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.

#### 4.3.3.3 Disabling Interrupts

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

#### 4.3.3.4 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 cancelled or reset based on the interrupt sensitivity ([Interrupt Sensitivity Level Programming](#)).

#### 4.3.3.5 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.

#### 4.3.3.6 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.

#### 4.3.3.7 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  $N-1$ . 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 4-1](#) 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 4-1 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 non-serializing, so the preceding sequence takes 4 clock cycles to execute.

#### 4.3.3.8 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 4-1](#) 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.

#### 4.3.3.9 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.

#### 4.3.3.10 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.

#### 4.3.3.11 Software Triggered Interrupt

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 mode 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.

#### 4.3.3.12 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.

#### 4.3.4 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.

##### 4.3.4.1 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.

##### 4.3.4.2 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 >= 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`.

#### 4.3.4.2.1 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`).

#### 4.3.4.2.2 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.

#### Example 4-2 Fast Interrupt Entry

```
EnterFastInterrupt(PI) {
    // Assert ((P != 0) && (STATUS32.IE) && (STATUS32.E ≥ 1)
    // && (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
    ILINK = PC;
    ERET = 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;
    // 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;}
    if (RGF_NUM_BANKS > 1){
        STATUS32.{RB=1}
    }
    AcknowledgeInterrupt(PI);
    Jump(InterruptVector(PI));
}
```

#### 4.3.4.2.3 Fast Interrupt Exit

The following example explains the sequence of a fast interrupt exit:

##### Example 4-3 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)) {
        //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);
    }
}
```

#### 4.3.4.3 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.

##### 4.3.4.3.1 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 3-3, 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; Hence, if AUX\_IRQ\_CTRL.U == 1, registers are pushed to the stack of the current user thread 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.

#### 4.3.4.3.2 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 prologue and epilogue 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 to restore 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 prologue or epilogue 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 prologue or epilogue sequence, after the exception is resolved and the handler executes the RTIE instruction, the failing prologue or epilogue 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 prologue or epilogue sequence, if the stack pointer is not 32-bit aligned, a misaligned access exception is always raised irrespective of the `STATUS32.AD` bit.

#### 4.3.4.3.3 Regular Interrupt Entry

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.

## Example 4-4 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 prologue 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 prologue
in_int_prologue = 0;
AcknowledgeInterrupt(PI);
Jump(InterruptVector(PI));
}
```



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](#).

**Table 4-4 Regular Interrupt Entry when SecureShield 2+2 Mode is Configured**

```

EnterInterrupt(PI, W) {
    /* Assert ((W < P) && (STATUS32.IE) && (W <= STATUS32.E)
       && (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
       indicates the prologue 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 prologue */
Check_handler_target(PI, AUX_IRQ_PRIORITY[PI].S)
in_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()
}
/* When -
sec_modes_option==true

```

**Table 4-4 Regular Interrupt Entry when SecureShield 2+2 Mode is Configured**

Functions used in the code are defined as follows:

```

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 prologue */
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 }
    }
}
}

```

**Table 4-4 Regular Interrupt Entry when SecureShield 2+2 Mode is Configured**

```

// 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)
    }
}

```

**Table 4-4 Regular Interrupt Entry when SecureShield 2+2 Mode is Configured**

```
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
}
```

#### 4.3.4.4 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.

#### 4.3.4.4.1 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.

### Example 4-5 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.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){
                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);
}
```

**Table 4-5 Regular Interrupt Exit when SecureShield 2+2 Modes 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.Z = STATUS32.U

        /* acknowledge interrupt */
        AcknowledgeInterrupt(PI)
        in_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_MPUs()
}

```

**Table 4-5 Regular Interrupt Exit when SecureShield 2+2 Modes is Configured**

```

/* UOP end, Processor start executing in mode of the fetching PC */
Set_mode_SID_by_MPUS()
}
/* 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, ILM.saved, 1)
    }
    /* if returning to secure user thread and secure kernel stack stored user
context */
    if ( (AUX_IRQ_ACT.ACTIVE == 0) && AUX_IRQ_ACT.U) {
        AUX_IRQ_ACT.U = 0
    }
    /* jump back to outer context */
    Check_jmp_target(IRM_saved, ILLINK)
    Jump(ILLINK)
/* UOP end, Processor start executing in mode of the fetching PC */
Set_mode_SID_by_MPUS()
}

```

**Table 4-5 Regular Interrupt Exit when SecureShield 2+2 Modes is Configured**

```

Functions used in the code are defined as follows:
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()
    }
}

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
}

```

**Table 4-5 Regular Interrupt Exit when SecureShield 2+2 Modes is Configured**

```

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)
            } else {
                Pop(Ri)
            }
        }
    }
    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
}

```



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](#).

#### 4.3.4.5 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 product 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

#### 4.3.4.6 Determine Active Interrupts

The following example demonstrates how to determine the active interrupts in a processor:

##### Example 4-6 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
}
```

## 4.4 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 4-7](#)). 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.

### 4.4.1 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 hence 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 hence a Memory Error 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.

A load or store operation that causes a bus error, and hence a [Bus Error Accessing External Instruction 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.

## 4.4.2 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

### 4.4.2.1 Vector Name

The vector name is a symbolic equivalent to the vector number.

### 4.4.2.2 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.

### 4.4.2.3 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 4-6](#).

**Table 4-6 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 interrupt vector table contain addresses of interrupt 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 interrupt vector table has undefined behavior.

If you need to explicitly pass control to an interrupt handler, use the AUX\_IRQ\_HINT register.

To pass control to an exception handler or perform a soft reset, do the following:

#### **Example 4-7 Pass Control to an Exception Handler or Perform a Soft Reset**

1. Read the contents of an interrupt 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 4-8 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
```

[Table 4-7](#) provides more information about the exception priorities and exception cause parameters.

#### **4.4.2.4 Cause Codes**

Because multiple exceptions share each vector, this 8-bit number is used to identify the exact cause of an exception.

#### **4.4.2.5 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.

#### **4.4.2.6 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 3-43](#) on page 147.

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*.

#### 4.4.3 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, Double Fault
3. Machine Check, Fatal TLB errors
  - a. Invalid TLB command
  - b. Overlapping TLB entries
  - c. Illegal Overlapping MPU entries
4. Interrupt
5. TLB Fault on Instruction Fetch
  - a. Machine Check, Memory Error on ITLB Access
  - b. ITLB miss
6. Protection Violation on Instruction Fetch (MMU or MPU)
7. Memory Error on instruction fetch:
  - a. Bus error accessing external instruction memory
  - b. Instruction fetch from unpopulated ICCM region
  - c. Instruction fetch spanning multiple instruction memory targets
8. Privilege Violation, Actionpoint hit on Instruction fetch
9. Machine Check, Internal Instruction Memory Error
10. Instruction Error
  - a. Illegal instruction exception
  - b. Illegal instruction sequence exception
11. Privilege Violation
  - a. Access to a secured resource in the normal mode
    - i. Kernel-only instruction or register violation
    - ii. Security-related instruction or register violation
  - a. Normal mode to secure mode return with the wrong return type
  - b. Normal mode to secure mode with unexpected user or kernel mode

- c. Interrupt or exception return fetch from the wrong mode
  - d. Access to a kernel resource in the user mode
  - e. SID violation on a secure resource
- b. APEX Extension Group – User mode access is disabled by XPU
  - c. APEX Extension Group – Access to privileged feature
    - i. Kernel-only Extension
    - ii. Attempt to access secure APEX feature from normal mode
    - iii. SID violation on access to APEX feature
- d. Attempt to halt a core in the normal mode
- 12. 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
  - 13. Extension Instruction Exception - requested by extension instruction
  - 14. Misaligned data memory access
  - 15. TLB miss on data access
    - a. Machine check, memory error on DTLB access
    - b. DTLB miss
  - 16. 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)
  - 17. Memory error on data access:
    - a. Data access to unpopulated DCCM region or absent memory
    - b. Data access spanning multiple data memory targets
  - 18. Machine check, internal data memory error
  - 19. Divide by zero exception when STATUS32[DZ] = 1
  - 20. Trap or software interrupt (TRAP\_S, SWI or SWI\_S instructions)
  - 21. Bus error on data access to external data memory (or ICCM) – an imprecise exception
  - 22. Actionpoint hit on Auxiliary register or Memory-access – an imprecise exception
  - 23. Machine check - uncorrectable ECC or parity error in vector memory
  - 24. Misaligned vector memory access
  - 25. Vector stack pointer checking protection violation

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 4-7](#) describes the exception vectors, exception cause codes, and exception parameters.

#### 4.4.3.1 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

#### 4.4.3.2 Exception Vectors and Cause Codes



[Table 4-7](#) indicates the ECR values when an exception is raised. If simultaneous violations occur, the Parameter field in ECR will have one bit set for each of those exceptions. For example, if there are simultaneous Code Protection, Stack Checking memory write violations, the Parameter field will read 3 (bit 0 is set for code protection and bit 1 is set for stack checking).

**Table 4-7 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(a)	Always
Bus Error from Data Memory	Memory Error	0x04	0x01	0x10	0x0u	0x01100u(a)'(b)	

**Table 4-7 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 from unpopulated ICCM region	Memory Error	0x04	0x01	0x01	0x00	0x010100	
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 from unpopulated DCCM region	Memory Error	0x04	0x01	0x11	0x00	0x011100	
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	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 4-7 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	
Double Fault-- exception accesses NS vector table in S memory (SecureShield only)	EV_MachineCheck	0x0C	0x03	0x00	0x01	0x030001	
Double Fault-- NS exception handler located in S memory (SecureShield only)	EV_MachineCheck	0x0C	0x03	0x00	0x02	0x030002	
Overlapping TLB Entries	EV_MachineCheck	0x0C	0x03	0x01	0x00	0x030100	
Fatal TLB Error	EV_MachineCheck	0x0C	0x03	0x02	0x00	0x030200	
Fatal Cache Error	EV_MachineCheck	0x0C	0x03	0x03	0x00	0x030300	
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 node 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
Secure Interrupt Programming Error	EV_MachineCheck	0x0C	0x03	0x14	0x00	0x031400	

**Table 4-7 Exception Vectors and Cause Codes (Continued)**

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value	Secure Mode
			Vector Number	Cause Code	Parameter		
Machine check - uncorrectable ECC or parity error in vector memory	EV_MachineCheck	0x0C	0x03	0x80	0x00	0x038000	
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 MMU(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
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 MMU (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

**Table 4-7 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 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	
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in MMU (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	

**Table 4-7 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 (EX) protection violation in MPU scheme (parameter code 0x04)	EV_ProtV	0x18	0x06	0x03	0x04	0x060304	
Memory Read-Modify-Write (EX) protection violation in MMU 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
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

**Table 4-7 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 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
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(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	
Data cache consistency error	EV_DCError	0x30	0x0C	0x00	0x00	0x0C0000	
Misaligned data access	EV_Misaligned	0x34	0x0D	0x00	0x00	0x0D0000	
Misaligned vector memory access	EV_VecUnit	0x38	0x0E	0x02	0x00	0x0E0200	
Vector stack pointer checking protection violation	EV_VecUnit	0x38	0x0E	0x05	0x00	0x0E0500	
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 243](#).
- (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.

#### 4.4.3.3 Reset

A Reset is an external reset signal that causes the ARCv2-based processor to perform a *hard* Reset. Upon Reset, various internal states of the ARCv2-based processor are pre-set 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. On 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. On 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.

A soft reset (an indirect jump to the Reset vector) does *not* pre-set any of the internal states of the ARCv2-based processor. For information about how to do a soft reset, see "[Pass Control to an Exception Handler or Perform a Soft Reset](#)" on page [222](#).

#### 4.4.3.4 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.

#### 4.4.3.5 Machine Check, Fatal TLB Error

- Any fatal error in the TLB or its memories (such as an uncorrectable parity or ECC error). This 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.
- Machine Check, Overlapping TLB Entries: Multiple matches for an address lookup in the TLB.
- 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.

#### 4.4.3.6 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	<ul style="list-style-type: none"> <li>■ JL/JL_S&lt;.d&gt;</li> <li>■ J/J_S&lt;.d&gt;</li> <li>■ Jcc&lt;.d&gt;/JEQ_S</li> <li>■ leave_s [..., pcl]</li> <li>■ BL&lt;.cc&gt;&lt;.d&gt;</li> <li>■ BL_S</li> <li>■ B&lt;.d&gt;</li> <li>■ B_S</li> <li>■ B&lt;.cc&gt;&lt;.d&gt;</li> <li>■ Bcc_S</li> <li>■ BBIT0</li> <li>■ BBIT1</li> <li>■ DBNZ&lt;.d&gt;</li> </ul>
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 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.

#### 4.4.3.7 Memory Error on Instruction Fetch

##### 4.4.3.7.1 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, 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.

Bus error exceptions on external instruction memory accesses are always reported precisely.

##### 4.4.3.7.2 Memory Error Exception, Fetch from Unpopulated Instruction Memory

A memory error exception is raised when an attempt is made to execute an instruction fetched from an ICCM memory region that is beyond the upper bound of the actual ICCM memory.

##### 4.4.3.7.3 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.

#### 4.4.3.8 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.

#### 4.4.3.9 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

#### 4.4.3.10 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.

#### 4.4.3.11 Instruction Error

##### 4.4.3.11.1 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 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 in order to determine the condition, and therefore the operands of BRcc and BBITn are always accessed.

If an instruction specifies a LIMM source operand, and the instruction executes in a delay-slot context (that is, when STATUS32.DE is set to 1), the LIMM value is considered to be unpredictable. Such instructions do not raise an Illegal Instruction exception on the basis of the illegality of the LIMM value itself. However, they raise an Illegal Instruction Sequence exception due to the use of a LIMM 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 LIMM operand. Normally, an access to an unimplemented auxiliary register triggers an Illegal Instruction exception. However, a LIMM 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 LIMM operand in a delay slot context, as described in [Illegal Instruction Sequence](#) section.

#### 4.4.3.11.2 Vector DSP Illegal Instruction Exception Summary

- vvset\_el\_size instruction is used with an incorrect value.
- Instructions use element\_size that is not supported for that instruction.
- Odd register number used for a register pair.
- The accumulator instruction uses an accumulator register that has not been configured.
- Scalar instruction does not fit in the maximum size of super instruction.
- Instruction formats that exceed the maximum size of a super instruction.
- Using VK or VL formats for instructions that do not support these immediates excluding VVNOP.
- Use of opcodes without opcode assignment to an instruction.
- Use of instruction formats that are not compatible with the processor configuration.
- Use of vvcaddacc or vvcsuccacc instructions with ALU0 while execution slot 1 has another scheduled instruction; although an instruction of the VPRED or VTOP class is allowed.
- Use of the vvscan\_excl\_add instruction with incompatible combination of VEC\_CTRL.VMODE and element\_size.
- The vvld.inc instruction is used with the A and B field using the same vector register.
- The AEX instruction is used with the auxiliary register VEC\_RF\_DATA

#### 4.4.3.11.3 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)
  - 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.



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

#### 4.4.3.11.4 Vector DSP Illegal Instruction Sequence

- Multiple instructions in a super instruction have the same destination register.
- Structural hazard: instruction in an execution slot where it can not be executed.
- A super instruction in a branch delay slot must have size 32 bits.
- If a super instruction is used as last instruction in the body of a zero-overhead loop, then it must have size 32 bits.
- A super instruction cannot be the target of an EI\_S instruction, that is, a super instruction cannot be included in the EI\_S instruction table.

- At most one scalar or flag-setting result can be produced per super instruction.
- The number of scalar source operands used by vector instructions in a bundle must not be more than three.
- The following scalar instructions are prohibited in bundles: BRK, BRK\_S, SWI, TRAP\_S, SLEEP, WEVT, WLFC, FLAG, KFLAG, SR, LR, AEX, EI\_S, all APEX instructions and all instructions handled by the micro-op sequencer (ENTER\_S, LEAVE\_S and RTIE).
- A branch instruction in a bundle is not allowed to have a delay slot.
- There can be at most one predication instruction per super instruction bundle. Predication instructions are the instructions that start with vvp.
- A vector instruction that sets the scalar Z-flag is not allowed to be in a branch delay slot.
- If a vector instruction generates the Z-flag, then there cannot be a scalar instruction bundled with it in the same super instruction; the scalar issue slot must be empty.
- A vector instruction is not allowed to use the ILINK register (r29) as scalar source operand in a vector-scalar instruction.
- There can be at most one vvset\_el\_size instruction per super instruction.

#### 4.4.3.12 Privilege Violation

##### 4.4.3.12.1 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.
- Attempts to access secure resources in the normal mode.
- SID violation on resource accesses for APEX, UAFX, 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 mode 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 epilogue sequence, and the exception PC is the RTIE instruction. If this exception is taken on exception epilogue, double fault will be raised.

#### 4.4.3.12.2 Privilege Violation, Kernel Only Access

Kernel-only instruction, core register, or auxiliary register is accessed from the user mode.

#### 4.4.3.12.3 Privilege Violation, Disabled Extension

Disabled instruction or register is accessed. The parameter field (nn) gives the group number (0-31) of the disabled extension.

#### 4.4.3.13 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 exception(s) raised

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

#### 4.4.3.14 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](#).

#### 4.4.3.15 TLB Miss on Data Access

##### 4.4.3.15.1 Data TLB Miss

Data TLB miss caused by a LD, ST, PUSH\_S, POP\_S, EX, LLOCK, or SCOND instruction. TLB lookup cannot locate an entry for the supplied virtual address

##### 4.4.3.15.2 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.

#### 4.4.3.16 Instruction Fetch TLB Miss

An instruction fetch caused a TLB miss. TLB lookup cannot locate an entry for the supplied virtual address.

#### 4.4.3.17 Machine Check, Instruction Fetch Memory Error

A memory error is triggered by an instruction fetch. Memory errors triggered by incorrectly speculated accesses are ignored.

#### 4.4.3.18 Protection Violation, Data Access

Memory Read, memory write, memory read-modify-write operations without the execute permission set (MMU, MPU, code protection, or stack checking).

#### 4.4.3.19 Memory Error on Data Access

##### 4.4.3.19.1 Memory Error Exception, Access to Unpopulated Data Memory

A memory error exception is raised when a data read or write is attempted within a DCCM memory region, and any part of the memory location accessed is beyond the upper bound of the actual DCCM memory.

##### 4.4.3.19.2 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

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.

#### 4.4.3.20 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.

#### 4.4.3.21 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.

#### 4.4.3.22 Trap or Software Interrupt

##### 4.4.3.22.1 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.

#### 4.4.3.22.2 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.

#### 4.4.3.22.3 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 un-cached, 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 accesses, 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.

#### 4.4.3.23 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*.

#### 4.4.3.24 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.

#### 4.4.3.25 Machine Check - Uncorrectable ECC or Parity Error in Vector Memory

Double-bit ECC error or single-bit parity error in the vector memory.

#### 4.4.3.26 Misaligned Vector Memory Access

Address of a vector element is not aligned on the element boundary.

#### 4.4.3.27 Vector Stack Pointer Checking Protection Violation

Vector stack pointer check violation is detected in the vector unit.

#### 4.4.4 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.

#### 4.4.5 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.

#### 4.4.6 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 4-7](#).
6. When a memory access triggers an exception, the exception fault address register (EFA) is loaded with the 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 actionpoints, 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 the 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.

#### 4.4.6.1 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 (`sys_tf_halt_r`) from each core, allowing external hardware to become aware of the fault and take any necessary action.

#### 4.4.7 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 4-8](#).

In [Table 4-8](#), 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 4-8 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

**Table 4-8 Exception and Interrupt Exit Modes (Continued)**

<b>U</b>	<b>AE</b>	<b>AUX_IRQ_ACT</b>	<b>FIRQ_OPTION</b>	<b>Current Mode</b>	<b>RTIE Response</b>	<b>Link Registers Used</b>
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
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	ERET, ERSTATUS, ERBTA

- U, AE, and AUX\_IRQ\_ACT are all set to 0 for state changes from 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).

#### 4.4.8 Exceptions and Delay Slots

For the ARCV2-based processor, exceptions are supported for instructions in the delay slots of branches.

### Example 4-9 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 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 cancelled, such as the possibility of a 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, 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 boundary, it is possible that only the first MPU region is accessed. In such cases an Illegal Instruction Sequence exception may be raised rather than a privilege violation.

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, and the presence of any such operand raises an Illegal Instruction Sequence exception.

In such cases, the value assigned to BLINK, normally includes the size of the long-immediate operand in the calculation of the size of the delay-slot instruction. 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.

#### 4.4.9 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 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 ( $\text{ERET} + \text{emulated\_instruction\_size} = \text{LP\_END}$ , and  $\text{LP\_COUNT} > 1$ ) (see [Exception Return Address, ERET](#)).
  - $\text{ERET} + \text{emulated\_instruction\_size}$  for normal, linear execution (see [Exception Return Address, ERET](#))

#### 4.4.10 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.



## 5

# 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 assembler 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 6, “32-bit Instruction Formats Reference”](#) and [Chapter 7, “16-bit Instruction Formats Reference”](#), and full details of all aspects of each individual instruction can be found in [Chapter 8, “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 [71](#).

## 5.1 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.

### 5.1.1 Top-level Instruction Formats

[Table 5-1](#) 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 5-1 Top-level Formats of the ARCv2 Instruction Set**

<b>Format</b>	<b>Description</b>	<b>Examples</b>
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_MV_HREG	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 5-1 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 prologue, epilogue	LD_S / LDB_S / ST_S / STB_S / ADD_S / SUB_S / PUSH_S / POP_S ENTER_S / LEAVE_S
F16_GP_LD_ADDF 16_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 Part 3)
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	

### 5.1.2 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 in order to maximize code density. Other profiles may use those major opcodes in different ways, to encode domain-specific instructions.

### 5.1.3 Assignment of Instruction Formats to Major Opcodes

Table 5-2 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 5-2 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	ARC64 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		F32_GEN_OP64
0x0C	F16_LD_ADD_RR			
0x0D	F16_ADD_IMM			
0x0E	F16_OP_HREG			
0x0F	F16_GEN_OP			
0x10	F16_LD_WORD			

**Table 5-2 Assignment of top-level instruction formats to major opcodes in each instruction set profile**

Major opcode	Common formats	Profile-specific formats		
		Compact Profile	Vector DSP Profile	ARC64 Profile
0x11		F16_LD_BYTEx	F32_VEC1	F16_LD_BYTEx
0x12		F16_LD_HALF	F32_VEC2	F16_LD_HALF
0x13		F16_LDX_HALF	F32_VEC3	F16_LDX_HALF
0x14	F16_ST_WORD			
0x15		F16_ST_BYTEx	F32_VEC4	F16_ST_BYTEx
0x16		F16_ST_HALF	F32_VEC5	F16_ST_HALF
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	F16_BL

## 5.1.4 32-bit Instruction Formats

**Table 5-3** 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 5-3 Summary of 32-bit instruction formats**

32-bit Format Hierarchy			Major Opcode		Layout of Instruction Formats																			
(a)	(b)	(c)																						
F32_BRO	COND UCOND_FAR	> 0x0	S[10:1]										0	S[20:11]										N R S[24:21]
F32_BR1	BL COND UCOND_FAR BCC REG_REG REG_U6	> 0x1	S[10:2]										0 1 0	S[20:11]										N R S[24:21] 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 ST_W6	> 0x3	B[2:0]		S[7:0]										S8	B[5:3]		C[5:0]	D		aa	ZZ	0 1	
F32_GEN4	REG_REG REG_U6 REG_S12 COND_REG COND_U6 LD_REG SOP ZOP	> 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]		U[5:0]	S[5:0]	S[11:6]	C[5:0]	0 1 Q[4:0]
	REG_REG REG_U6	> 0x4	B[2:0]		00 01	1 0 1 1 1 1					F	B[5:3]		C[5:0]	i[5:0]		U[5:0]							
	REG_REG REG_U6	> 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		U[5:0]							
F32_EXT5, F32_EXT6, F32_APEX	REG_REG REG_U6 REG_S12 COND_REG COND_U6 SOP ZOP	> 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]		U[5:0]	S[5:0]	S[11:6]	C[5:0]	0 1 Q[4:0]
	REG_REG REG_U6	> 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)		U[5:0]							
	REG_REG REG_U6	> 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		U[5:0]							

## 5.1.5 16-bit Instruction Formats

Table 5-4 below summarizes the 16-bit compact instruction formats defined by the ARCv2 instruction set.

**Table 5-4 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]		1																
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																		
	SOP	>					i[7:5]		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]																						

## 5.1.6 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 361. [Table 5-16](#) on page 265 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 5-5](#) on page 258 and [Table 5-6](#) on page 259 list the notation used for the encoding.

**Table 5-5 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
D	Di	<.di> direct data cache bypass

**Table 5-5 Key for 32-bit Addressing Modes and Encoding Conventions (Continued)**

Encoding Character	Encoding Field	Syntax
A	A	<.aa> address writeback mode
Z	Z	<.zz> data size
X	X	<.x> sign extend
K	K	<.k> store constant operand mode

**Table 5-6 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 high (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 high (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)

### 5.1.7 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.

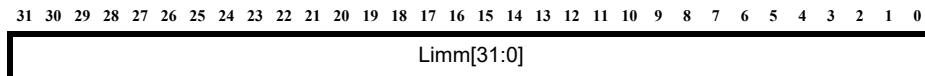
When a destination register field contains 62, the result of the instruction is discarded and no destination register is updated. Any status flag updates still occur according to the set-flags directive (.F) or if flag setting is implicit in the instruction.

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 please refer to the section on “[Instruction Layout in Memory](#)” on page [90](#).

**Figure 5-1 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)

### 5.1.8 Condition Code Tests

[Table 5-7](#) lists the codes used for condition code tests.

**Table 5-7 Condition Codes**

Code Q field	Mnemonic	Condition	Test
0x00	AL, RA	Always	1
0x01	EQ , Z	Zero	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)
0x0D	HI	Higher than (unsigned)	/C and /Z
0x0E	LS	Lower than or same (unsigned)	C or Z

**Table 5-7 Condition Codes (Continued)**

<b>Code Q field</b>	<b>Mnemonic</b>	<b>Condition</b>	<b>Test</b>
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.

### 5.1.9 Encoding Branch and Jump Delay Slot Modes

[Table 5-8](#) lists the codes used for delay slot modes on Branch and Jump instructions.

**Table 5-8 Delay Slot Modes**

<b>N Bit</b>	<b>Mode</b>	<b>Operation</b>
0		Only execute the next instruction when not jumping (default)
1	D	Always execute the next instruction

### 5.1.10 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.

#### 5.1.10.1 Assembler Syntax for Static Branch Predictions

[Table 5-9](#) lists the interpretation of the <T> assembler 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 5-9 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 5-9](#) 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 5-9](#) lists these cases in bold font.

### 5.1.11 Encoding Load / Store Address Write-back Modes

[Table 5-10](#) lists the codes used for address write-back modes in Load and Store instructions.

**Table 5-10 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 5-10 Address Write-Back Modes (Continued)**

<b>AA bits</b>	<b>Address Mode</b>	<b>Memory Address Used</b>	<b>Register Value Write-back</b>
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 <a href="#">Table 5-13</a> scaling_shift is defined in <a href="#">Table 5-11</a> .	No write-back

**Table 5-11 Scaling Shift**

<b>ZZ Code</b>	<b>ZZ Suffix</b>	<b>scaling_shift</b>
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.

### 5.1.12 Encoding Load / Store Cache Bypass Mode

[Table 5-12](#) lists the codes used for direct to memory bypass modes in Load and Store instructions.

**Table 5-12 Cache Bypass Modes**

<b>Di bit</b>	<b>Di Suffix</b>	<b>Access Mode</b>
0		Default access to memory. Cached data memory access ( <i>default, if no &lt;.di&gt; field syntax</i> )
1	DI	Direct to memory, bypassing data-cache (if available). Non-cached data memory access

### 5.1.13 Encoding Load / Store Data Sizes

[Table 5-13](#) lists the codes used for data size modes in Load and Store instructions.

**Table 5-13 Load Store Data Sizes**

<b>ZZ Code</b>	<b>ZZ Suffix</b>	<b>Access Mode</b>
00		Word (32-bit), default behavior
01	B	Byte

**Table 5-13 Load Store Data Sizes (Continued)**

<b>ZZ Code</b>	<b>ZZ Suffix</b>	<b>Access Mode</b>
10	H	Half-word (16-bit)
11	D	Raises an <a href="#">Illegal Instruction</a> exception if LL64_OPTION is disabled, otherwise indicates double-word data.

### 5.1.14 Encoding Load Data Extension Modes

[Table 5-14](#) lists the codes used to encode extension semantics in Load instructions.

**Table 5-14 Load Data Extend Mode**

<b>X bit</b>	<b>X Suffix</b>	<b>Extension Semantics</b>
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

### 5.1.15 Encoding Store Data Source Mode

[Table 5-15](#) 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 5-15 Store Data Source Mode**

<b>K bit</b>	<b>K Suffix</b>	<b>Store Data Source</b>
0		The ‘c’ operand field (Bits 6 thru 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.

### 5.1.16 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.

## 5.1.17 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

### 5.1.17.1 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.

### 5.1.17.2 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.

## 5.2 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, as shown below:

- |      |                              |
|------|------------------------------|
| 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, the

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 8-1](#) on page [362](#). The following notation is used to describe the syntax of instructions.

**Table 5-16 Instruction Syntax Convention**

a	Destination register (reduced range for 16-bit instruction.)	
---	--	--

**Table 5-16 Instruction Syntax Convention (Continued)**

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	
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	
Z	Zero flag	
N	Negative flag	
C	Carry flag	
V	Overflow flag	
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 writeback	
<.d>	Optional delay slot mode	
<.di>	Optional direct data cache bypass	
<.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	

## 5.3 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.

### 5.3.1 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.

### 5.3.2 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

## 5.4 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 instruction 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 [259](#)).

### 5.4.1 Integer Adder Operations

[Table 5-17](#) summarizes the instructions that perform integer addition, subtraction or comparison operations. These instructions are always supported in all ARCv2 processors.

**Table 5-17 Integer addition, subtraction and comparison operations**

Instruction	Operation	Description
<a href="#">ADD</a>	$a \leftarrow b + c$	add
<a href="#">ADC</a>	$a \leftarrow b + c + C$	add with carry
<a href="#">SUB</a>	$a \leftarrow b - c$	subtract
<a href="#">SBC</a>	$a \leftarrow (b - c) - C$	subtract with carry
<a href="#">CMP</a>	$b - c$	compare
<a href="#">RCMP</a>	$c - b$	reverse compare
<a href="#">RSUB</a>	$a \leftarrow c - b$	reverse subtract
<a href="#">ADD1</a>	$a \leftarrow b + (c \ll 1)$	add with left shift by 1
<a href="#">ADD2</a>	$a \leftarrow b + (c \ll 2)$	add with left shift by 2
<a href="#">ADD3</a>	$a \leftarrow b + (c \ll 3)$	add with left shift by 3
<a href="#">SUB1</a>	$a \leftarrow b - (c \ll 1)$	subtract with left shift by 1
<a href="#">SUB2</a>	$a \leftarrow b - (c \ll 2)$	subtract with left shift by 2
<a href="#">SUB3</a>	$a \leftarrow b - (c \ll 3)$	subtract with left shift by 3

### 5.4.1.1 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.

### 5.4.1.2 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.

### 5.4.1.3 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, i.e. 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.

## 5.4.2 Move Operations

**Table 5-17** 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 5-18 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
SEXBXW	$a \leftarrow \text{sexb } (b)$	Sign-extend byte to word
SEXH	$a \leftarrow \text{sexh } (b)$	Sign-extend half to word

### 5.4.2.1 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 8, “Instruction Set Details”](#).

#### 5.4.2.2 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 8, “Instruction Set Details”](#).

#### 5.4.2.3 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 8, “Instruction Set Details”](#).

### 5.4.3 Bit-wise Logical Operations

[Table 5-19](#) summarizes the group of instructions that perform bit-wise logical operations. These instructions are always supported in all ARCv2 processors.

**Table 5-19 Bit-wise logical operations**

Instruction	Operation	Description
<a href="#">AND</a>	$a \leftarrow b \text{ and } c$	logical bit-wise AND
<a href="#">OR</a>	$a \leftarrow b \text{ or } c$	logical bit-wise OR
<a href="#">BIC</a>	$a \leftarrow b \text{ and not } c$	logical bit-wise AND with invert
<a href="#">XOR</a>	$a \leftarrow b \text{ exclusive-or } c$	logical bit-wise exclusive-OR
<a href="#">TST</a>	$b \text{ and } c$	bit-wise test, updating flags only
<a href="#">NOT</a>	$a \leftarrow \sim b$	bit-wise negation

## 5.4.4 Bit Operations and Mask Operations

### 5.4.5 Shift and Rotate Operations

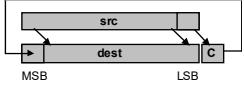
Table 5-20 Bit-mask operations

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

Table 5-21 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

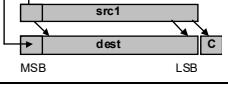
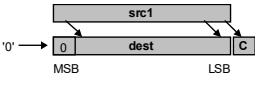
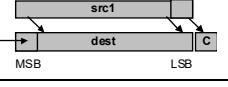
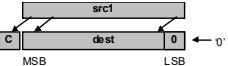
**Table 5-21 Single Bit Shift and Rotate Operations**

Instruction	Operation	Description
<b>RRC</b>		Rotate right through carry

**5.4.5.1 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 5-21](#) on page [271](#) lists the single bit shift instructions provided as single operand instructions.

**Table 5-22 Multi-bit Shift Operations**

Instruction	Operation	Description
<b>ASR multiple</b>		Multiple arithmetic shift right, sign filled
<b>LSR multiple</b>		Multiple logical shift right, zero filled
<b>ROR multiple</b>		Multiple rotate right
<b>ASL Multiple</b>		Multiple arithmetic shift left, zero filled
<b>XBFU</b>	$a \leftarrow (b >> c[4:0]) \text{ and } ((1 << (c[9:5]+1))-1)$	Extract unsigned bit-field

**Table 5-23 Shift-assist operations**

Instruction	Operation	Description
<b>LSL16</b>	$a \leftarrow b \text{ asl } 16$	Arithmetic shift left 16 places
<b>LSR16</b>	$a \leftarrow b \text{ lsr } 16$	Logical shift right 16 places

**Table 5-23 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

#### 5.4.6 Selection Operations

**Table 5-24 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

#### 5.4.7 Byte-swapping Operations

**Table 5-25 Byte-swapping operations**

Instruction	Operation	Description
SWAP	$b \leftarrow \text{swap}(c)$	Swap words
SWAPE	$b \leftarrow \text{swap\_endian}(c)$	Swap endianness

#### 5.4.8 Bit-scanning Operations

**Table 5-26 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

### 5.4.9 Relational Comparison Operations

**Table 5-27 Relational Comparison Operations**

<b>SETcc variant</b>	<b>Operation</b>	<b>Description</b>
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	$a \leftarrow (b >= c)$ signed	Set if greater than or equal
SETLO	$a \leftarrow (b < c)$ unsigned	Set if lower than
SETHS	$a \leftarrow (b >= c)$ 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

### 5.4.10 Integer Multiply, Multiply-accumulate, and Divide Operations

**Table 5-28 Integer Multiply, MAC and Divide Operations**

<b>Instruction</b>	<b>Operation</b>	<b>Description</b>
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
MAC	$acc \leftarrow acc + (b * c)$ $a \leftarrow acc\_lo$	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	$acc \leftarrow acc + (b * c)$ $a \leftarrow acc\_lo$	32 x 32 signed integer multiply-accumulate, returning 64 bits of the accumulated results
MACDU	$acc \leftarrow acc + (b * c)$ $a \leftarrow acc\_lo$	32 x 32 unsigned integer multiply-accumulate, returning 64 bits of the accumulated results

**Table 5-28 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

### 5.4.11 Dual and Quad Integer Multiply / Accumulate Operations

**Table 5-29 Dual and Quad Integer Multiply and MAC Operations**

Instruction	Operation	Description
DMPYH	<pre>if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}</pre>	Dual 16x16 multiply
DMPYHU	<pre>if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}</pre>	Dual 16x16 multiply unsigned
DMACH	<pre>if (cc) {     result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));     a = result.w0;     acc = result; }</pre>	Dual 16x16 MAC
DMACHU	<pre>if (cc) {     result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));     a = result.w0;     acc = result; }</pre>	Dual 16x16 MAC unsigned

### 5.4.12 Integer Vector Operations

**Table 5-30 Integer Vector ADD, SUB, MPY operations**

Instruction	Operation	Description
VADD2H	<pre>if (cc) {     a.h0 = b.h0 + c.h0;     a.h1 = b.h1 + c.h1; }</pre>	Vector-add, dual 16-bit
VSUB2H	<pre>if (cc) {     a.h0 = b.h0 - c.h0;     a.h1 = b.h1 - c.h1; }</pre>	Vector-sub, dual 16-bit

**Table 5-30 Integer Vector ADD, SUB, MPY operations**

<b>Instruction</b>	<b>Operation</b>	<b>Description</b>
<b>VADDSSUB2H</b>	if (cc) { a.h0 = b.h0 + c.h0; a.h1 = b.h1 - c.h1; }	Vector-add/sub, dual 16-bit
<b>VSUBADD2H</b>	if (cc) { a.h0 = b.h0 - c.h0; a.h1 = b.h1 + c.h1; }	Vector-sub/add, dual 16-bit
<b>VMPY2H</b>	if (cc) { acc.w0 = A.w0 = b.h0 * c.h0; acc.w1 = A.w1 = b.h1 * c.h1; }	Vector-multiply, dual 16x16
<b>VMPY2HU</b>	if (cc) { acc.w0 = A.w0 = b.h0 * c.h0; acc.w1 = A.w1 = b.h1 * c.h1; }	Unsigned vector-multiply, dual 16x16

#### 5.4.13 Special and Synchronizing Instructions

**Table 5-31 Special / Synchronizing Instructions**

<b>Instruction</b>	<b>Operation</b>	<b>Description</b>
SLEEP	Sleep	Sleep
SWI	Software Interrupt	Software interrupt or breakpoint
SYNC	Synchronize	Wait for all data-based memory transactions to complete
RTIE	Return	Return from interrupt/exception
BRK	Breakpoint	Breakpoint instruction
SETI	Set Interrupt Enable	Set or restore interrupt enable and priority
CLRI	Clear Interrupt Enable	Clear interrupt enable, optionally save interrupt state
WEVT	Enter sleep mode	Enter Sleep State to reduce dynamic power during busy-waiting loops
WLFC	Enter sleep mode and wait on event	Enter sleep mode and wait on event
DSYNC	Synchronize instruction	Wait for completion of all outstanding data memory transactions

**Table 5-31 Special / Synchronizing Instructions (Continued)**

Instruction	Operation	Description
DMB	Data memory barrier	Wait for completion of selected type of data memory operations before initiating similar types of data memory operations
FLAG	aux.reg[] $\leftarrow$ c	Set status flags

## 5.5 Memory Access Instructions

These instructions include data transfer operations from and to Auxiliary registers, memory locations, and data transformation instructions such as sign-extend and rotate instructions.

**Table 5-32 Memory Access Instructions**

Instruction	Operation	Description
EX	b $\leftarrow$ mem[c]; mem[c] b	Atomic Exchange
LD	Load register-register	See section <a href="#">General Operations Register-register Format</a> on page 318
LDI	b $\leftarrow$ memory[LDI_BASE + (src1 << 2)]	Load Indexed
LLOCK	b $\leftarrow$ mem[c]; LF $\leftarrow$ 1;	Load locked
SCOND	if LF mem[c] $\diamond$ b	Store conditional
LLOCKD	B $\leftarrow$ mem[C]; LF $\leftarrow$ 1;	Load locked on 64-bit data
SCONDD	if LF mem[C] $\diamond$ 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 suffices: .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.

### 5.5.1 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 <i>ld a,[b,0]</i> )
LD<zz><.x><.aa><.di>	a,[b,s9]	
LD<zz><.x><.di>	a,[limm,s9]	(Redundant format, use <i>ld a,[limm]</i> )
LD<zz><.x><.di>	a,[limm]	(= <i>ld a,[limm,0]</i> )
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]	( <i>u7 offset is 32-bit aligned, reduced set of regs</i> )
LD_S	r0, [GP, s11]	( <i>s11 offset is 32-bit aligned, reduced set of regs</i> )
LDB_S	r0, [GP, s9]	( <i>reduced set of regs</i> )
LDH_S	r0, [GP, s10]	( <i>s10 offset is 16-bit aligned, reduced set of regs</i> )
LD_S	b, [PCL, u10]	( <i>u10 offset is 32-bit aligned, reduced set of regs</i> )
LD_S	r, [h, u5]	( <i>r is one of r0-r3, u5 offset is 32-bit aligned, reduced set of regs</i> )
LD_S.AS	a, [b, c]	( <i>reduced set of regs</i> )
LD_S	r1, [GP,s9]	( <i>reduced set of regs</i> )

## 5.5.2 Store Instructions

### 5.5.2.1 Store Register with Offset

The following is the syntax for the Store register ([ST STH STB STD](#)) 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]	( <i>u7 offset is 32-bit aligned</i> )
STB_S	b, [SP, u7]	( <i>u7 offset is 32-bit aligned</i> )
ST_S	c, [b, u7]	( <i>u7 offset is 32-bit aligned</i> )
STB_S	c, [b, u5]	
STH_S	c, [b, u6]	( <i>u6 offset is 16-bit aligned</i> )

## 5.5.3 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] (*Push c onto the stack*)

PUSH\_S b       *(Push b onto the stack)*  
PUSH\_S BLINK   *(Push BLINK onto the stack)*

The following is the instructions syntax for pop ([POP\\_S](#)) operations on the stack:

LD.AB a,[SP,+4]   *(Pop top item of stack to a)*  
POP\_S b           *(Pop top item of stack to b)*  
POP\_S BLINK      *(Pop top item of stack to BLINK)*

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.

#### 5.5.4 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.

Two forms 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 EX instruction exchanges the contents of the specified memory location with the contents of the specified 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.

The ARCv2 architecture also provides special Load-locked and Store-conditional instructions, to enable the implementation of lock-free data structures. 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. The check succeeds only if the lock flag is still set, and if the SCOND and LLOCK addresses match. Any interrupt or exception clears the lock flag, and any external write to the lock location will also clear 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.

### 5.5.5 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. Please refer to 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.
- PREALLOC: Allocates a cache line as a preparation for writing the entire line. The line is not fetched from the memory subsystem.

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 5-33 PREFETCH Instruction Encodings**

Class	<code>&lt;d&gt;</code>	<code>&lt;x&gt;</code>	Instruction
PREFETCH	0	0	PREFETCH
PREFETCH	0	1	PREALLOC
PREFETCH	1	0	PREFETCHW

**Table 5-33 PREFETCH Instruction Encodings**

<b>Class</b>	<b>&lt;d&gt;</b>	<b>&lt;x&gt;</b>	<b>Instruction</b>
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 5-34](#). These combinations cover all load and prefetch-type instructions in 32-bit encodings.

**Table 5-34 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions**

<b>Class</b>	<b>&lt;d&gt;</b>	<b>&lt;x&gt;</b>	<b>&lt;aa&gt;</b>	<b>&lt;zz&gt;</b>	<b>Instruction</b>
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

## 5.6 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 5-35 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

### 5.6.1 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 318](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 319](#), and the [General Operations Register with Signed 12-bit Immediate](#) format on [page 319](#).

### 5.6.2 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 318](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 319](#), and the [General Operations Register with Signed 12-bit Immediate](#) format on [page 319](#).

### 5.6.3 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 318](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 319](#), the [General Operations Register with Signed 12-bit Immediate](#) format on [page 319](#), and the [General Operations Conditional Register with Unsigned 6-bit Immediate](#) format on [page 320](#).

## 5.7 Control Flow Instructions

These instructions include the Branch, jump, conditional, and interrupt-related operations.

**Table 5-36 Control Flow Operations**

Instruction	Operation	Description
J	$\text{pc} \leftarrow c$	Unconditional jump
Jcc	$\text{if } (\text{cc}) \text{ pc} \leftarrow c$	Conditional jump
JL	$\text{blink} \leftarrow \text{next\_pc};$ $\text{pc} \leftarrow c$	Jump and link
JLcc	$\text{if } (\text{cc}) \{$ $\text{blink} \leftarrow \text{next\_pc};$ $\text{pc} \leftarrow c$ }	Jump and link conditionally
JLI_S	$\text{blink} \leftarrow \text{next\_pc};$ $\text{pc} \leftarrow \text{JLI\_BASE} + (\text{u10} \ll 2);$	Jump and link indexed
EI_S	$\text{BTA} \leftarrow \text{next\_pc};$ $\text{STATUS32.ES} \leftarrow 1$ $\text{pc} \leftarrow \text{EI\_BASE} + (\text{u10} \ll 2);$	Execute indexed
B B_S	$\text{pc} \leftarrow \text{pcl} + \text{offset}$	Branch unconditionally
Bcc Bcc_S	$\text{if } (\text{cc}) \{$ $\text{pc} \leftarrow \text{pcl} + \text{offset}$ }	Branch conditionally
BLcc	$\text{if } (\text{cc}) \{$ $\text{blink} \leftarrow \text{next\_pc};$ $\text{pc} \leftarrow \text{pcl} + \text{offset}$ }	Branch-and-link conditionally
BRcc	$\text{if } (\text{cmp}(b,c)) \{$ $\text{pc} \leftarrow \text{pcl} + \text{offset}$ }	Compare-and-branch
BBIT0	$\text{if } (b[c] == 0) \{$ $\text{pc} \leftarrow \text{pcl} + \text{offset}$ }	Branch if bit is zero
BBIT1	$\text{if } (b[c] == 1) \{$ $\text{pc} \leftarrow \text{pcl} + \text{offset}$ }	Branch if bit is one
BI	$\text{pc} \leftarrow \text{next\_pc} + (\text{c} \ll 2)$	Branch indexed

**Table 5-36 Control Flow Operations**

Instruction	Operation	Description
BIH	$pc \leftarrow next\_pc + (c \ll 1)$	Branch indexed half-word
LPcc	$LP\_END \leftarrow pcl + s13$ $LP\_START \leftarrow next\_pc$	Unconditional loop setup (16-bit aligned target address)
LPcc	<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 prologue 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 epilogue sequence	Performs a programmable sequence of operations to restore callee-saved registers, deallocate a stack frame, and return to the caller.

### 5.7.1 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 in order 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 5-37](#).

**Table 5-37 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 5-7](#) on page [260](#) lists the condition codes that are available for conditional branch instructions.

### 5.7.1.1 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

### 5.7.1.2 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 [608](#)).

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*)

### 5.7.1.3 Branch On Compare or Bit Test

Branch on Compare (BRcc) and Branch on Bit Test (BBIT0, BBIT1) have a branch range of  $\pm 256B$ . 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 5-38](#) lists the additional condition code tests that are available through the effect of reversing the operands.

**Table 5-38 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 5-39 Branch on Compare Pseudo Mnemonics, Register-Register**

Mnemonic	Condition
BRGT b,c,s9	Branch if b-c is greater than (encode as BRLT c,b,s9)
BRLE b,c,s9	Branch if b-c is less than or equal (encode as BRGE c,b,s9)
BRHI b,c,s9	Branch if b-c is higher than (encode as BRLO c,b,s9)
BRLS b,c,s9	Branch if b-c is lower than or same (encode as BRHS c,b,s9)

[Table 5-40](#) 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 5-40 Branch on Compare Pseudo Mnemonics, Register-Immediate**

Mnemonic	Condition
BRGT b,u6,s9	Branch if b-u6 is greater than (encode as BRGE b,u6+1,s9)
BRLE b,u6,s9	Branch if b-u6 is less than or equal (encode as BRLT b,u6+1,s9)

**Table 5-40 Branch on Compare Pseudo Mnemonics, Register-Immediate (Continued)**

Mnemonic	Condition
BRHI b,u6,s9	Branch if b-u6 is higher than (encode as BRHS b,u6+1,s9)
BRLS b,u6,s9	Branch if b-u6 is lower than or same (encode as BRLO b,u6+1,s9)

## 5.7.2 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 5-41](#).

**Table 5-41 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 \(r31\)](#)).

## 5.7.3 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.

### 5.7.4 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.

## 5.8 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 5-42 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
RTIE	Return from Interrupt or Exception	Return from interrupt/exception
SYNC	Synchronize with memory	Synchronize with memory

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
```

### 5.8.1 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.

### 5.8.2 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.

### 5.8.3 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.

### 5.8.4 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
```

### 5.8.5 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
```

### 5.8.6 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.

### 5.8.7 Synchronization Instructions

The synchronize instruction, SYNC, waits until all data-based memory operations (LD, ST, cache fills) have completed.

## 5.9 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.

### 5.9.1 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
```

## 5.9.2 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
```

## 5.9.3 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
```

## 5.10 Floating Point Instructions

**Table 5-43 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	DMULH11	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 5-43 Floating-Point Unit Instructions, 0x06**

<b>Sub-opcode B field (6 bits)</b>	<b>Instruction</b>	<b>F Bit</b>	<b>Description</b>
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

#### 5.10.0.1 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
-------------	----------

### 5.10.0.2 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

## 5.11 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 5-44 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 writeback
D	32x32 with 64-bit writeback
F	Fractional operand
T	24-bit operands
R	Rounding
U	Unsigned
C	Complex conjugate

**Table 5-44 Instruction Mnemonics**

L	Least significant
M	Most significant

### 5.11.1 Basic Saturating Arithmetic Operations

**Table 5-45 Basic Saturating Arithmetic Operations**

Instruction	Type	Major	Minor	Description
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
ASRCSR	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

## 5.11.2 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 82. Short immediates are replicated for each vector lane.

**Table 5-46 Vector Unpacking Operations**

Instruction	Type	Major	Minor	F bit	Description
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
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
VALGN2H	DOP	0x05	0x0D	0	Two-way 16-bit vector align

## 5.11.3 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 82. Short immediates are replicated for each vector lane.

**Table 5-47 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

**Table 5-47 Vector ALU Operations**

<b>Instruction</b>	<b>Type</b>	<b>Major</b>	<b>Minor</b>	<b>F bit</b>	<b>Description</b>
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
VMAC2H	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
VMIN2H	DOP	0x05	0x25	1	Two way vector minimum 16 bits

#### 5.11.4 Accumulator Operations

**Table 5-48 Accumulator Operations**

<b>Instruction</b>	<b>Type</b>	<b>Major</b>	<b>Minor</b>	<b>F bit</b>	<b>Description</b>
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

### 5.11.5 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 82. Short immediates are replicated for each vector lane.

**Table 5-49 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
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
VMPY2HU	DOP	0x05	0x20	0	Two way vector 16 bit saturating fractional multiplication; two way 32 bit write back

### 5.11.6 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 [82](#). Short immediates are replicated for each vector lane.

**Table 5-50 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

### 5.11.7 32x32 and 16x16 MAC Operations

**Table 5-51 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

**Table 5-51 32x32 and 16x16 MAC Operations**

<b>Instruction</b>	<b>Type</b>	<b>Major</b>	<b>Minor</b>	<b>F bit</b>	<b>Description</b>
<b>MPYW</b>	DOP	0x04	0x1E	F	16 bit signed integer multiplication
<b>MPYUW</b>	DOP	0x04	0x1F	F	16 bit unsigned integer multiplication
<b>MAC</b>	DOP	0x05	0x0E	F	32 bit signed integer multiplication and accumulation
<b>MACU</b>	DOP	0x05	0x0F	F	32 bit unsigned integer multiplication and accumulation
<b>MPYD</b>	DOP	0x05	0x18	F	32 bit signed integer multiplication; 64-bit result
<b>MPYDU</b>	DOP	0x05	0x19	F	32 bit unsigned integer multiplication; 64-bit result
<b>MACD</b>	DOP	0x05	0x1A	F	32 bit signed integer multiplication and accumulation; 64-bit result
<b>MACDU</b>	DOP	0x05	0x1B	F	32 bit unsigned integer multiplication and accumulation; 64-bit result
<b>MPYM MPYH</b>	DOP	0x06	0x0A	F	32 bit signed fractional saturating multiplication
<b>MPYFR</b>	DOP	0x06	0x0B	F	32 bit signed fractional saturating an rounding multiplication
<b>MACU</b>	DOP	0x06	0x0C	F	32 bit signed fractional saturating multiplication and accumulation
<b>MACFR</b>	DOP	0x06	0x0D	F	32 bit signed fractional saturating an rounding multiplication and accumulation
<b>MSUBF</b>	DOP	0x06	0x0E	F	32 bit fractional saturating subtraction
<b>MSUBFR</b>	DOP	0x06	0x0F	F	32 bit fractional saturating and rounding subtraction
<b>MPYDU</b>	DOP	0x06	0x12	F	32 bit signed fractional saturating multiplication; 64-bit result
<b>MACDU</b>	DOP	0x06	0x13	F	32 bit signed fractional saturating multiplication and accumulation; 64-bit result
<b>MSUBDF</b>	DOP	0x06	0x15	F	32 bit signed fractional saturating subtraction; 64-bit result
<b>MPYW_S</b>	DOP	0x0F	0x09	F	16 bit signed integer multiplication

**Table 5-51 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

### 5.11.8 Dual 16x8 MAC Operations



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 [82](#). Short immediates are replicated for each vector lane.

**Table 5-52 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

### 5.11.9 32x16 MAC Operations

**Table 5-53 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
MACWHUL	DOP	0x06	0x1F	F	32x16 bit unsigned multiplication and accumulation, lower signed 16 bits

**Table 5-53 32x16 MAC Operations**

<b>Instruction</b>	<b>Type</b>	<b>Major</b>	<b>Minor</b>	<b>F bit</b>	<b>Description</b>
<b>MPYWHFM</b>	DOP	0x06	0x20	F	32x16 bit signed fractional multiplication, higher signed 16 bits
<b>MPYWHFMR</b>	DOP	0x06	0x21	F	32x16 bit signed fractional multiplication with rounding, higher signed 16 bits
<b>MACWHFM</b>	DOP	0x06	0x22	F	32x16 bit signed fractional multiplication and accumulation, higher signed 16 bits
<b>MACWHFMR</b>	DOP	0x06	0x23	F	32x16 bit signed fractional multiplication and accumulation with rounding, higher signed 16 bits
<b>MACWHFL</b>	DOP	0x06	0x24	F	32x16 bit signed fractional multiplication, lower signed 16 bits
<b>MPYWHFL</b>	DOP	0x06	0x25	F	32x16 bit signed fractional multiplication with rounding, lower signed 16 bits
<b>MPYWHFLR</b>	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.
<b>MACWHFLR</b>	DOP	0x06	0x27	F	Signed 32x16 fractional multiplication and accumulation with rounding.
<b>MACWHKL</b>	DOP	0x06	0x28	F	Signed 32x16 integer multiplication and accumulation and shift-right the lower bits
<b>MACWHKUL</b>	DOP	0x06	0x29	F	Unsigned 32x16 integer multiplication and accumulation and shift-right the lower bits
<b>MPYWHKL</b>	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.
<b>MPYWHKUL</b>	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.
<b>MSUBWHFL</b>	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.

**Table 5-53 32x16 MAC Operations**

<b>Instruction</b>	<b>Type</b>	<b>Major</b>	<b>Minor</b>	<b>F bit</b>	<b>Description</b>
<b>MSUBWHFLR</b>	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.
<b>MSUBWHFMR</b>	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.
<b>MSUBWHFMR</b>	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

### 5.11.10 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 [82](#). Short immediates are replicated for each vector lane.

**Table 5-54 Complex Operations**

<b>Instruction</b>	<b>Type</b>	<b>Major</b>	<b>Minor</b>	<b>F bit</b>	<b>Description</b>
<b>CBFLYHF1R</b>	SOP	0x06	0x19	0	16 + 16 bit complex FFT butterfly, second half;
<b>CMPYHNFR</b>	DOP	0x06	0x00	1	16 + 16 bit complex signed fractional multiplication with rounding, no fractional shift
<b>CMPYHFR</b>	DOP	0x06	0x01	1	16 + 16 bit complex signed fractional multiplication with rounding
<b>CMPYCHNFR</b>	DOP	0x06	0x02	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding
<b>CMPYCHFR</b>	DOP	0x06	0x05	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding
<b>CMACHNFR</b>	DOP	0x06	0x06	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding with no fractional shift

**Table 5-54 Complex Operations**

<b>Instruction</b>	<b>Type</b>	<b>Major</b>	<b>Minor</b>	<b>F bit</b>	<b>Description</b>
CMACHFR	DOP	0x06	0x07	1	16 + 16 bit complex signed fractional multiplication and accumulation with rounding
C MACCHNFR	DOP	0x06	0x08	1	16 + 16 bit complex conjugated signed fractional multiplication and accumulation with rounding with no fractional shift
C MACCHFR	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



# 6

## 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 [251](#) lists and summarizes the 32-bit formats. In particular, [Table 5-4](#) on page [257](#) 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 5-16](#) on page [265](#), and section “[Encoding Notation](#)” on page [258](#) 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, please refer to [Chapter 5, “Instruction Set Summary”](#).

For full details on the available encodings and syntax for each instruction please refer to the relevant page for each instructions in [Chapter 8, “Instruction Set Details”](#).

## 6.1 Branch Format [F32\_BR0]

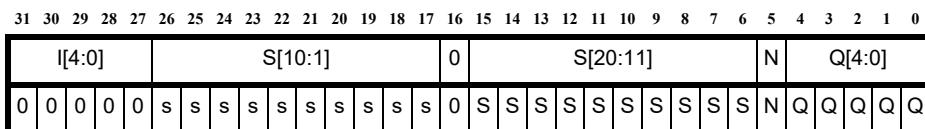
The F32\_BR0 format is used to encode branch instructions with large displacements. There are two sub-formats, COND and UCOND\_FAR. The COND format is used to encode conditional branch instructions, with an optional delay slot, and the UCOND\_FAR format is used to encode unconditional branches with maximum 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.

### 6.1.1 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 5-7](#) on page [260](#). For information about the encoding of delay-slot modes, see [Table 5-8](#) on page [261](#).

**Figure 6-1 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.

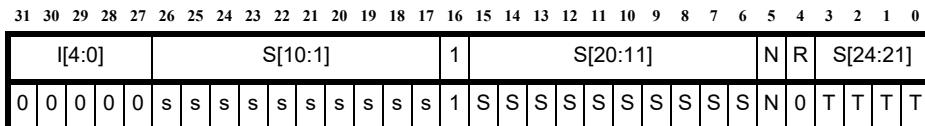
Syntax:

Bcc<.d> s21 (branch if condition is true)

### 6.1.2 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 5-8](#) on page [261](#) for information on delay slot mode encoding.

**Figure 6-2 Branch Unconditional Far Format**



Syntax:

B<.d> s25 (unconditional branch far)

## 6.2 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 5-8](#) on page [261](#).

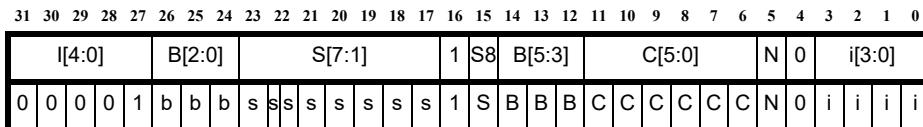
These instructions also provide a way to encode a static branch prediction. This is a hint to hardware that, in the absence of a dynamic prediction, the branch is statistically more likely to go one way than the other. Not all implementations of the ARCV2 ISA exploit these hints, but compilers should 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 below in "[Static Branch Prediction Mode](#)" on page [312](#)

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).

### 6.2.1 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 6-3 Branch on Compare Register-register Format**



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)*  
 BBIT1<.d><.T> b,c,s9     *(branch if bit c in reg b is set)*

### 6.2.2 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 6-4 Branch on Compare Register-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]				S[7:1]				1	S8	B[5:3]				U[5:0]				N	1	i[3:0]							
0	0	0	0	1	b	b	b	s	s	s	s	s	s	1	S	B	B	B	U	U	U	U	U	U	N	1	i	i	i		

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*)

### 6.2.2.1 Static Branch Prediction Mode

Table 5-9 on page 262 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 will typically give acceptable results on the majority of branches.

### 6.2.2.2 Encoding of BRcc, BBIT0, and BBIT1 Instructions

**Table 6-1 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)
0x05	BRHS	src1 – src2	BTFN	src1 higher than or same as src2 (unsigned)
0x06	BBIT0	$(src1 \text{ and } 2^{src2}) == 0$	FTBN	bit src2 in src1 is clear
0x07	BBIT1	$(src1 \text{ and } 2^{src2}) == 0$	FTBN	bit src2 in src1 is set
0x08	BREQ	src1 – src2	FTBN	src1 is equal to src2

**Table 6-1 Branch on Compare/Bit Test Register-Register (Continued)**

<b>Sub-opcode</b>			<b>Static Branch Prediction</b>	<b>Branch Condition Tested</b>
<b>i field (4 bits)</b>	<b>Instruction</b>	<b>Operation</b>		
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	$(src1 \text{ and } 2^{src2}) == 0$	BTFN	bit src2 in src1 is clear
0x0F	BBIT1	$(src1 \text{ and } 2^{src2}) == 0$	BTFN	bit src2 in src1 is set

### 6.2.3 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.

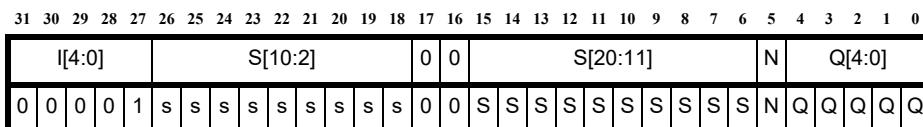
All function entry points are aligned to a 4-byte boundary, requiring that the target address is also 4-byte aligned. Thus the least-significant two bits of the branch offset can be assumed to be zero, and are omitted from the instruction encoding.

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 5-8](#).

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.

#### 6.2.3.1 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 5-7](#) on page 260.

**Figure 6-5 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.

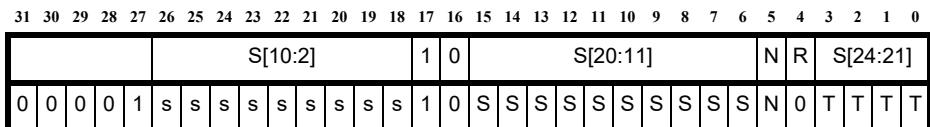
Syntax:

BLcc<.d> s21 (*branch if condition is true*)

### 6.2.3.2 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 6-6 Branch and Link Unconditional Far Format**



The reserved field, R, is ignored.

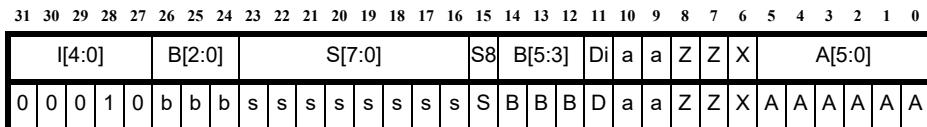
Syntax:

BL<.d> s25 (*unconditional branch far*)

## 6.3 Load Register with Offset Format [F32\_LD\_OFFSET]

For information about encoding the Load instruction, see [Table 5-10](#) on page [262](#), [Table 5-11](#) on page [263](#), [Table 5-13](#) on page [263](#), and [Table 5-14](#) on page [264](#).

**Figure 6-7 Load Register with Offset Format**



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 Id a,[limm])
LD<zz><.x><.di>      a,[limm]     (= Id a,[limm,0])
```

## 6.4 Store Register with Offset Format [F32\_ST\_OFFSET]

For information about encoding the Store instruction, see [Table 5-10](#) on page [262](#), [Table 5-13](#) on page [263](#), and [Table 5-13](#) on page [263](#).

**Figure 6-8 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	B	B	B	C	C	C	C	C	D	a	a	Z	Z	0		

**Figure 6-9 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]	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	B	B	B	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]
```

## 6.5 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 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.

### 6.5.1 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 6-2** 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 6-2 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	i[5:0] except when (i[5:0] == 101111)	F	B[5:3]	C[5:0]	A[5:0]	U[5:0]	S[5:0]	S[11:6]	C[5:0]	0	Q[4:0]	U[5:0]	1	C[5:0]	F	B[5:3]	C[5:0]	i[5:0]	(except 111111)	101111	F	i[5:3]	C[5:0]	U[5:0]	111111			
		REG_U6	>			01																												
		REG_S12	>			10																												
		COND_REG	>			11																												
		COND_U6	>																															
	SOP	REG_REG	>	F32_GEN4	B[2:0]	00	101111	F	B[5:3]	C[5:0]	i[5:0]	(except 111111)	U[5:0]	C[5:0]	F	B[5:3]	C[5:0]	i[5:0]	101111	F	i[5:3]	C[5:0]	U[5:0]	111111	F	i[5:3]	C[5:0]	U[5:0]	111111					
		REG_U6	>			01																												
	ZOP	REG_REG	>	F32_GEN4	i[2:0]	00	101111	F	i[5:3]	C[5:0]	i[5:0]	111111	F	i[5:3]	C[5:0]	F	B[5:3]	C[5:0]	i[5:0]	111111	F	i[5:3]	C[5:0]	U[5:0]	111111	F	i[5:3]	C[5:0]	U[5:0]	111111				
		REG_U6	>			01																												

### 6.5.2 Operand Format Indicators

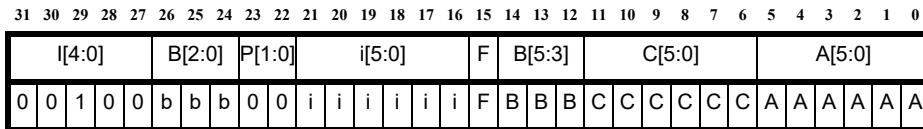
The General Operations formats provide five operand sub-formats, encoded with the P[1:0] and M fields, located in instruction bits [23:22] and [5] respectively, as shown in [Table 6-3](#).

**Table 6-3** 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

### 6.5.3 Syntax and Encoding of Instructions in General Operations Formats

#### 6.5.3.1 General Operations Register-register Format

**Figure 6-10** 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 319*)
- 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*)

### 6.5.3.2 General Operations Register with Unsigned 6-bit Immediate

**Figure 6-11 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	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`

### 6.5.3.3 General Operations Register with Signed 12-bit Immediate

**Figure 6-12 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		

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*)

### 6.5.3.4 General Operations Conditional Register

**Figure 6-13 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]							
0	0	1	0	0	b	b	b	1	0	i	i	i	i	i	i	F	B	B	B	C	C	C	C	C	M	Q	Q	Q	Q		

**Figure 6-13 General Operations Conditional Register Format**

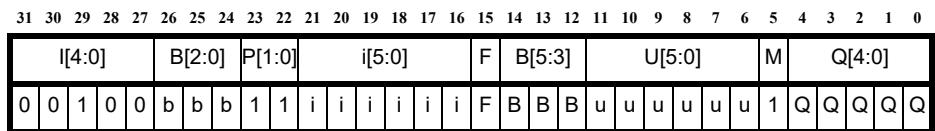
A value of 0x2F in the first operator field, i, indicates a SOP or ZOP format instruction, which is invalid for this operand mode and raises an [Illegal Instruction](#) exception.

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)
```

#### 6.5.3.5 General Operations Conditional Register with Unsigned 6-bit Immediate

**Figure 6-14 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)
```

## 6.5.4 Dual-operand Instructions, F32\_GEN4

The DOP opcode assignments for F32\_GEN4 are shown below in [Table 6-4](#). Subsequent sections contain similar tables for SOP and ZOP format instructions, and for other top-level formats (e.g. 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 <.F> bit set to 0, and the second row of the pair encodes instructions for which the <.F> bit is 1.

Any attempt to execute an instruction with an opcode marked **Illegal** in one of the opcode tables, will raise 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 will not result in an exception. These cases indicate that the <.F> bit is ignored during the decoding of the instruction that is positioned directly above the **Reserved** cell.

**Table 6-4      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	BTST	SUB3	J.D	FLAG	LD	SETNE
	1	ADC.F	MIN.F	BTST.F	SUB3.F	Reserved	KFLAG		SETNE.F
010	0	SUB	MOV	BXOR	MPYLO	JL	LR	LD	SETLT
	1	SUB.F	MOV.F	BXOR.F	MPYLO.F	Reserved	Reserved		SETLT.F
011	0	SBC	Illegal	BMSK	MPYHI	JL.D	SR	LD	SETGE
	1	SBC.F	TST	BMSK.F	MPYHI.F	Reserved	Reserved		SETGE.F
100	0	AND	DBNZ	ADD1	MPYHIU	BI	BMSKN	LD	SETLO
	1	AND.F	CMP	ADD1.F	MPYHIU.F	Illegal	BMSKN.F		SETLO.F
101	0	OR	DBNZ.D	ADD2	MPYLOU	BIH	XBFU	LD	SETHS
	1	OR.F	RCMP	ADD2.F	MPYLOU.F	Illegal	XBFU.F		SETHS.F
110	0	BIC	RSUB	ADD3	MPYW.F	LDI	CRC	LD	SETLE
	1	BIC.F	RSUB.F	ADD3.F	MPYW.F	Reserved	CRC.F		SETLE.F
111	0	XOR	BSET	SUB1	MPYWU.F	AEX	SOP_FMT	SETGT	SETGT
	1	XOR.F	BSET.F	SUB1.F	MPYWU.F	Reserved		SETGT.F	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 6-5 Special DOP sub-formats in F32\_GEN4**

Special-format instructions		Major Opcode		Layout of Instruction Formats																															
Instruction	i[5:0]																																		
				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																															
J<cc>	0x20			0x4	R	0 0 0 1 1 0 1 1	i[5:0]					C[5:0]					Reserved																		
J<cc>.D	0x21											U[5:0]																							
JL<cc>	0x22											S[5:0]					S[11:6]																		
JL<cc>.D	0x23											C[5:0]					0	Q[4:0]																	
												U[5:0]																							
				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																															
				0x4	R	0 1 1 0 1 1	i[5:0]					U[5:0]					Reserved																		
												S[5:0]					S[11:6]																		
												U[5:0]					1	Q[4:0]																	
				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																															
				0x4	B[2:0]	0 0 0 1 1 0 1 1	i[5:0]					C[5:0]					Reserved																		
												U[5:0]																							
												S[5:0]					S[11:6]																		
												C[5:0]					0	Q[4:0]																	
				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																															
				0x4	R	0 0 0 1 1 0 1 1	i[5:0]					C[5:0]					Reserved																		
												U[5:0]					Reserved																		
												S[5:0]					S[11:6]																		
												C[5:0]					0	Q[4:0]																	
				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																															
				0x4	B[2:0]	0 0 0 1 1 0 1 1	i[5:0]					C[5:0]					Reserved																		
												U[5:0]					Reserved																		
												S[5:0]					S[11:6]																		
												C[5:0]					0	Q[4:0]																	
				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																															
				0x4	R	0 0 0 1 1 0 1 1	i[5:0]					C[5:0]					Reserved																		
												U[5:0]					Reserved																		
												S[5:0]					S[11:6]																		
												C[5:0]					0	Q[4:0]																	
				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																															
				0x4	B[2:0]	0 0 0 1 1 0 1 1	i[5:0]					C[5:0]					Reserved																		
												U[5:0]					Reserved																		
												S[5:0]					S[11:6]																		
												C[5:0]					0	Q[4:0]																	
				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																															
				0x4	R	0 0 0 1 1 0 1 1	i[5:0]					C[5:0]					Reserved																		
												U[5:0]					Reserved																		
												S[5:0]					S[11:6]																		
												C[5:0]					0	Q[4:0]																	
				31 30 29 28 27 26 25 24 23 2																															

### 6.5.5 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.

### 6.5.6 Jump and Jump-and-Link Conditionally, 0x04, [0x20 – 0x23]

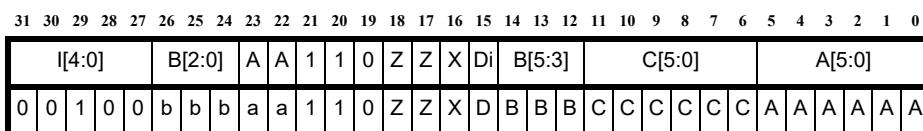
The jump (Jcc) and jump-and link (JLcc) instructions are specially encoded in major opcode 0x04 in which the B field is reserved and must be set to 0x0. Any value in the B field is ignored by the processor. The destination register, A field, must also be set to 0x0 when the operand mode, P, is 0x0 or 0x1. If P is 0x0 or 0x1, any value in the A field is ignored.

### 6.5.7 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 5-10](#) on page [262](#), [Table 5-14](#) on page [264](#), and [Table 5-13](#) on page [263](#).

**Figure 6-15 Load Register-Register Format**



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,imm]

LD<zz><.x><.di> a,[imm,c]

### 6.5.8 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 6-6    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	

### 6.5.9 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 6-7    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	Illegal	Illegal	Illegal	Illegal	Illegal
		Illegal							
001	0	SLEEP	WLFC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal							
010	0	SWI	DSYNC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal							

**Table 6-7    Opcode assignment for ZOP instructions in major op-code 0x04 (F32\_GEN4)**

011	0	SYNC	DMB	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal							
100	0	RTIE	Illegal						
	1	Illegal							
101	0	BRK	Illegal						
	1	Illegal							
110	0	SETI	Illegal						
	1	Illegal							
111	0	CLRI	Illegal						
	1	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 6-3 on page 318*)

SETI    c

SETI    u6

CLRI    c

CLRI    u6

## 6.5.10 Dual-operand Instructions, F32\_EXT5

**Table 6-8** 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
000	0	ASL	REM	DMPYH	MPYD	VMPY2HWF	DMPYHWF	QMPYH	VADD4H
	1	ASL.F	REM.F	DMPY.F	MPYD.F	SJLI	DMPYHWF.F	QMPYH.F	Illegal
001	0	LSR	REMU	DMPYHU	MPYDU	VASL2H	VPACK2HL	QMPYHU	VSUB4H
	1	LSR.F	REMU.F	DMPYHU.F	MPYDU.F	VASLS2H	VPACK2HM	QMPYHU.F	Illegal
010	0	ASR	ASLS	DMACH	MACD	VASR2H	DMPYHF	DMPYWH	VADDSUB4H
	1	ASR.F	ASLS.F	DMACH.F	MACD.F	VASRS2H	DMPYHF.F	DMPYWH.H	Illegal
011	0	ROR	ASRS	DMACHU	MACDU	VLSR2H	DMPYHFR	DMPYWHU	VSUBADD4H
	1	ROR.F	ASRS.F	DMACHU.F	MACDU.F	VASRSR2H	DMPYHFR.F	DMPYWHU.F	Illegal
100	0	DIV	ASRSR	VADD2H	VMPY2H	VADD4B	DMACHF	QMACH	VADD2
	1	DIV.F	ASRSR.F	VADDS2H	VMPY2HF	VMAX2H	DMACHF.F	QMACH.F	Illegal
101	0	DIVU	VALIGN2H	VSUB2H	VMPY2HU	VSUB4B	DMACHFR	QMACHU	VSUB2
	1	DIVU.F	SETACC	VSUBS2H	VMPY2HFR	VMIN2H	DMACHFR.F	QMACHU.F	Reserved
110	0	ADDS	MAC	VADDSUB2H	VMACH2H	ADCS	VPERM	DMACWH	VADDSUB
	1	ADDS.F	MAC.F	VADDSUBS2H	VMAC2HF	ADCS.F	BSPUSH	DMACWH.F	Illegal
111	0	SUBS	MACU	VSUBADD2H	VMACH2HU	SBCS	SOP_FMT	DMACWHU	VSUBADD
	1	SUBS.F	MACU.F	VSUBADDSS2H	VMAC2HFR	SBCS.F		DMACWHU.F	Illegal

## 6.5.11 Single-operand Instructions, F32\_EXT5

**Table 6-9** 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	FFS	GETACC	VEXT2BHLF	VABS2H	SQRT	Illegal
		SWAP.F	NORMH.F	FFS.F	Illegal	Illegal	Illegal	SQRT.F	Illegal
001	0	NORM	SWAPE	FLS	NORMACC	VEXT2BHMF	VABSS2H	SQRTF	Illegal
	1	NORM.F	SWAPE.F	FLS.F	Illegal	Illegal	Illegal	SQRTF.F	Illegal
010	0	SATH	LSL16	Illegal	SATF	VREP2H	VNEG2H	Illegal	Illegal
	1	SATH.F	LSL16.F	Illegal	SATF.F	Illegal	Illegal	Illegal	Illegal
011	0	RNDH	LSR16	Illegal	Illegal	VREP2HM	VNEGS2H	Illegal	Illegal
	1	RNDH.F	LSR16.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	ABSSH	ASR16	Illegal	VPACK2HBL	VEXT2BHL	VNORM2H	Illegal	Illegal
	1	ABSSH.F	ASR16.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

**Table 6-9** Opcode assignment for SOP instructions in major op-code 0x05 (F32\_EXT5)

101	0	ABSS	LSR8	Illegal	VPACK2HBM	VEXT2BHM	Illegal	Illegal	Illegal
	1	ABSS.F	LSR8.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	NEGHS	ROL8	Illegal	VPACK2HBLF	VSEXT2BHL	BSPEEK	Illegal	Illegal
	1	NEGH.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	NEGS	ROR8	Illegal	VPACK2HBMF	VSEXT2BHM	PSPOP	Illegal	ZOP_FMT
	1	NEGS.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	

**6.5.12 Zero-operand Instructions, F32\_EXT5****Table 6-10** 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						
	1	Illegal							
001	0	ASLSACC	Illegal						
	1	Illegal							
010	0	Illegal							
	1	Illegal							
011	0	Illegal							
	1	Illegal							
100	0	FLAGACC	Illegal						
	1	Illegal							
101	0	MODIF	Illegal						
	1	Illegal							
110	0	Illegal							
	1	Illegal							
111	0	Illegal							
	1	Illegal							

**6.5.13 Dual-operand Instructions, F32\_EXT6****Table 6-11** 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	MACWHLK	FDMUL	FCVT64
	1	CMPYHNFR	CMACCHNFR	DIVF.F	DMACHBL.F	MPYWHFM.F	MACWHLK.F	Illegal	Illegal

**Table 6-11 Opcode assignment for DOP instructions in major op-code 0x06 (F32\_EXT6)**

001	0	FSADD	FCVT32_64	VMAC2HNFR	DMACHBM	MPYWHFMR	MACWHKUL	FDADD	FCVT64_32
	1	CMPYHFR	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	CMPYHFMR	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

### 6.5.14 Single-operand Instructions, F32\_EXT6

**Table 6-12 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
		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
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	

## 6.5.15 Zero-operand Instructions, F32\_EXT6

**Table 6-13 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							
001	0	Illegal							
	1	Illegal							
010	0	Illegal							
	1	Illegal							
011	0	Illegal							
	1	Illegal							
100	0	Illegal							
	1	Illegal							
101	0	Illegal							
	1	Illegal							
110	0	Illegal							
	1	Illegal							
111	0	Illegal							
	1	Illegal							

## 6.6 APEX Extension Instruction Format [F32\_APEX]

The F32\_APEXtop-level format, assigned to major opcode 0x07, is available for use by User Extension Instructions defined using the APEX extension mechanism.

Figure 6-16 using major opcode 0x05 as an example to show the syntax of op<.f> a,b,c encoding.

**Figure 6-16** 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	A	A	A	A	A	A	

### **6.6.1 Extension ALU Operation, Register with Unsigned 6-bit Immediate**

Figure 6-17 shows the APEX encoding syntax of op<.f> a,b,u6 encoding.

**Figure 6-17 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	0	1	b	b	b	0	1	i	i	i	i	i	i	F	B	B	B	u	u	u	u	u	A	A	A	A	A	A	

### **6.6.2 Extension ALU Operation, Register with Signed 12-bit Immediate**

Using major opcode 0x05 as an example, Figure 6-18 shows the syntax of op<.f> b,b,s12 encoding.

**Figure 6-18 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	0	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	

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.

### 6.6.3 Extension ALU Operation, Conditional Register

Figure 6-19 shows the syntax of op<.cc><.f> b,b,c that is encoded.

**Figure 6-19 Extension ALU Operation, conditional register**

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.

#### 6.6.4 Extension ALU Operation, Conditional Reg with Unsigned 6-bit Immediate

Using major opcode 0x05 as an example, [Figure 6-20](#) shows the syntax of op<.cc><.f> b,b,u6 encoding.

**Figure 6-20 Extension ALU Operation, cc 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]			M	Q[4:0]																			
0	0	1	0	1	b	b	b	1	1	i	i	i	i	i	i	f	B	B	B	u	u	u	u	u	u	1	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, raise an [Illegal Instruction](#) exception unless the Q field specifies a condition code defined as an extension condition.

#### 6.6.5 FastMath Extension Pack Instructions

[Table 6-14](#) lists the FastMath Pack Extension Instructions. If this extension is available with your processor, the instructions will be available in the *Part: FastMath Pack Extension* in the *ARCV2 ISA Programmer's Reference Manual*.

**Table 6-14 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 abs(X) < abs(Y) or X = -Y, and Y != 0.
FMP_DIVF15	Fractional division of Q15 numerator X, by Q15 divisor Y, where abs(X) < abs(Y) or X = -Y, and Y != 0.
FMP_RECIP	Computes the reciprocal of a Q31 fraction X, provided X != 1.
FMP_RECIP15	Computes the reciprocal of a Q15 fraction X, provided X != 1.
FMP_SQRTF	Computes the square root of the Q31 fractional argument X, when X >= 0.
FMP_SQRTF15	Computes the square root of the Q15 fractional argument X, when X >= 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 6-14 List of FastMath extension pack instructions**

<b>Instruction</b>	<b>Operation</b>
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 = log2(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 = log2(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).

## 6.6.6 CryptoPack Instructions

[Table 6-15](#) lists the 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.

**Table 6-15 List of CryptoPack Extension Instructions**

<b>Instruction</b>	<b>Operation</b>
sha256_sigma0	Computes the $\Sigma 0$ operation.
sha256_sigma1	Computes the $\Sigma 1$ operation.
sha256_gamma01	Computes the $\Gamma 0 + \Gamma 1$ operation for the message scheduler.

**Table 6-15 List of CryptoPack Extension Instructions**

<b>Instruction</b>	<b>Operation</b>
sha256_upd	Returns the updated hash value at the end of the 64 rounds.
sha256_rnd	Applies a full round transformation to the internal state.
aes_byte	Extract the desired byte from a 32-bit word.
aes_rxor8	Right rotates one byte operand2 and bitwise xor it with operand1.
aes_rxor16	Right rotates two bytes operand2 and bitwise xor it with operand1.
aes_rxor24	Right rotates three bytes operand2 and bitwise xor it with operand1.
aes_qRoundF	Perform a quarter of the first round for encryption or decryption.
aes_qRoundN	Perform a quarter of an intermediate round for encryption or decryption.
aes_qRoundL	Perform a quarter of the last round for encryption or decryption.
aes_sboxword	Substitutes each byte of the input with the S-box or invS-box.
aes_mixword	Performs the linear transformation MixColumns or InvMixcolumns.
pk_madd	Perform one multiply and add operation with carry.
pk_maddset	Set the multiplicative constant, return current value of carry and reset it.
pk_mac96	Perform one multiply and 96-bit add operation with accumulate.
pk_macs96	Perform one multiply, shift, and 96-bit add operation with accumulate.
pk_forw96	Set up for next loop iteration.
des_ebox1	Generate first 16 bits of DES E permutation.
des_ebox2	Generate last 32 bits of DES E permutation.
des_fbox1	Generate first 32 bits of DES final permutation.
des_fbox2	Generate last 32 bits of DES final permutation.
des_ibox1	Generate first 32 bits of DES initial permutation.
des_ibox2	Generate last 32 bits of DES initial permutation.
des_pbox	Generate the DES P permutation.
des_sbox	Generate the 48-bit to 32-bit Sbox permutation.



# 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 5-4](#) on page [257](#) 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 5-16](#) on page [265](#), and section “[Encoding Notation](#)” on page [258](#) 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, please refer to [Chapter 5, "Instruction Set Summary"](#).

For full details on the available encodings and syntax for each instruction please refer to the relevant page for each instructions in [Chapter 8, "Instruction Set Details"](#).

## 7.1 Compact Move/Load [F16\_COMPACT]

### Extension Group: BASELINE

**Figure 7-1 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 7-2 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	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.

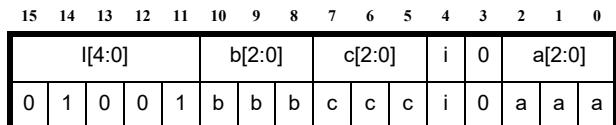
Syntax:

LD\_S R0/1/2/3, [h,u5]

## 7.2 Compact Load/Add/Sub [F16\_LD\_ADD\_SUB]

Extension Group: CODE\_DENSITY

**Figure 7-3 Compact Load and Subtract Format**



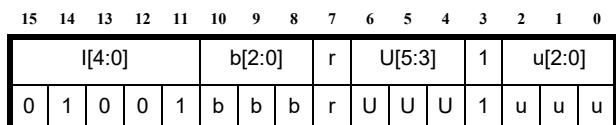
The first sub-format of opcode 0x09 encodes additional LD\_S.AS and SUB\_S instructions in a compact format to improve code density.

Syntax:

LD\_S.AS      a, [b, c]

SUB\_S      a, b, c

**Figure 7-4 Compact Add Format**



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.

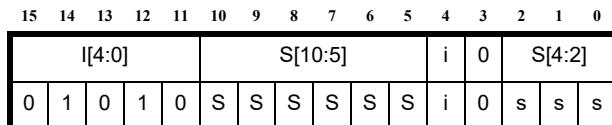
Syntax:

ADD\_S      r, b, u6    r can be R0 or R1

## 7.3 Compact Load/Store [F16\_LD\_ST\_1]

Extension Group: CODE\_DENSITY

**Figure 7-5 Compact Load and Store Format**



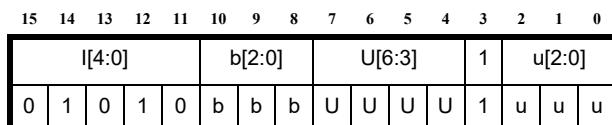
The first sub-format in opcode 0x0A encodes two additional GP-relative load/store instructions for use when accessing the most frequently occurring small data objects.

Syntax:

LD\_S R1, [GP, s11]

ST\_S R0, [GP, s11]

**Figure 7-6 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 LIMM 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.

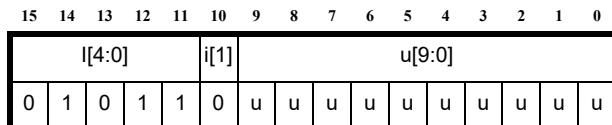
Syntax:

LDI\_S b, [u7]

## 7.4 Indexed Jump or Execute [F16\_JLI\_EI]

Extension Group: CODE\_DENSITY

**Figure 7-7 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:

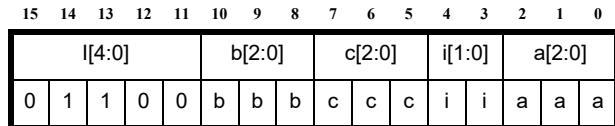
EI\_S u10

**Table 7-1 16-Bit Indexed Jump and Link Operations**

Sub-opcode i field (1 bit)	Instruction	Operation	Description
0x0	JLI_S	BLINK $\leftarrow$ next PC PC $\leftarrow$ JLI_BASE + (u10 << 2)	Jump and link, to indexed table location
0x1	EI_S	STATUS32.ES $\leftarrow$ 1 BTA $\leftarrow$ next PC PC $\leftarrow$ EI_BASE + (u10 << 2)	Execute instruction at table index

## 7.5 Load /Add Register-Register [F16\_LD\_ADD\_RR]

**Figure 7-8 Load and Add Register-Register Format**



Syntax:

LD\_S a, [b, c]

LDB\_S a, [b, c]

LDH\_S a, [b, c]

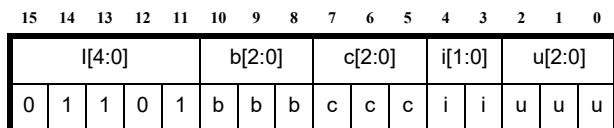
ADD\_S a, b, c

**Table 7-2 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

## 7.6 Add/Sub/Shift Register-Immediate [F16\_ADD\_IMM]

**Figure 7-9 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 7-3 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.

## 7.7 Dual Register Operations [F16\_OP\_HREG]

**Figure 7-10 Dual Register with a 3-bit b Register Format**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
l[4:0]					b[2:0]			h[2:0]			i[1:0]		h[4:3]		
0	1	1	1	0	b	b	b	h	h	h	i	i	0	H	H

**Figure 7-11 Dual Register with a 3-bit Biased Signed Integer Format**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
l[4:0]					s[2:0]			h[2:0]			i[1:0]		h[4:3]		
0	1	1	1	0	s	s	s	h	h	h	i	i	1	H	H

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 7-4 16-Bit MOV/CMP/ADD with High Register**

<b>Sub-opcode</b> <b>i field (3 bits)</b>	<b>Instruction</b>	<b>Operation</b>	<b>Description</b>
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

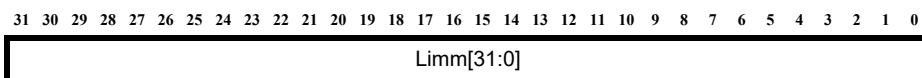
### 7.7.1 Long Immediate Operands in Format 0x0E

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 7-12 Long Immediate Operand Format**

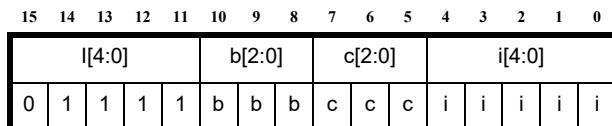
Syntax:

limm

## 7.8 General Register Format Instructions [F16\_GEN\_OP]

### 7.8.1 DOP-format General Operations

**Figure 7-13 Compact General Operations Register-Register Format**



Syntax:

op\_S b,b,c

op\_S b,c

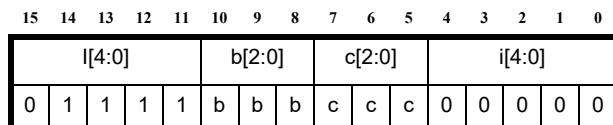
**Table 7-5 DOP\_format 16-Bit General Operations**

Sub-opcode i field (5 bits)	Instruction	Operation	Description
0x00	SOPs	c field is sub-opcode2	See <a href="#">Table 7-6</a> on page 347.
0x01		<b>Illegal Instruction</b>	Reserved
0x02	SUB_S	b $\leftarrow$ b - c	Subtract
0x03		<b>Illegal Instruction</b>	Reserved
0x04	AND_S	b $\leftarrow$ b and c	Logical bitwise AND
0x05	OR_S	b $\leftarrow$ b or c	Logical bitwise OR
0x06	BIC_S	b $\leftarrow$ b and not c	Logical bitwise AND with invert
0x07	XOR_S	b $\leftarrow$ b exclusive-or c	Logical bitwise exclusive-OR
0x08		<b>Illegal Instruction</b>	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 and c	Test
0x0C	MPY_S	b $\leftarrow$ b * c	32 X 32 Multiply (least-sig. half)
0x0D	SEXBX_S	b $\leftarrow$ sexb c	Sign extend byte
0x0E	SEXH_S	b $\leftarrow$ sexw c	Sign extend 16-bit half-word

**Table 7-5 DOP\_format 16-Bit General Operations (Continued)**

<b>Sub-opcode</b> <b>i field (5 bits)</b>	<b>Instruction</b>	<b>Operation</b>	<b>Description</b>
0x0F	EXTB_S	b ← extb c	Zero extend byte
0x10	EXTH_S	b ← extw c	Zero extend 16-bit half-word
0x11	ABS_S	b ← abs c	Absolute
0x12	NOT_S	b ← not c	Logical NOT
0x13	NEG_S	b ← neg c	Negate
0x14	ADD1_S	b ← b + (c << 1)	Add with left shift by 1
0x15	ADD2_S	b ← b + (c << 2)	Add with left shift by 2
0x16	ADD3_S	b ← b + (c << 3)	Add with left shift by 3
0x17		Illegal Instruction	Reserved
0x18	ASL_S	b ← b asl c	Multiple arithmetic shift left
0x19	LSR_S	b ← b lsr c	Multiple logical shift right
0x1A	ASR_S	b ← b asr c	Multiple arithmetic shift right
0x1B	ASL_S	b ← c + c	Arithmetic shift left by one
0x1C	ASR_S	b ← c asr 1	Arithmetic shift right by one
0x1D	LSR_S	b ← c lsr 1	Logical shift right by one
0x1E	TRAP_S	Trap	Raise Exception
0x1F	BRK_S	Break	Break (Encoding is 0x7FFF)

### 7.8.2 SOP-format 16-bit General Operations

**Figure 7-14 Compact Single Operand, Jumps, Special Formats**

Syntax:

op\_S      b

op\_S      b,b

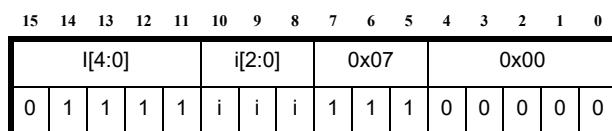
J\_S<.d> [b]  
 JL\_S<.d> [b]  
 SUB\_S.ne b,b,b

**Table 7-6 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 <a href="#">Table 7-7</a> on page 348.

### 7.8.3 ZOP-format 16-bit General Operations

#### Compact Zero Operand Instructions



Syntax:

```
op_S
NOP_S
UNIMP_S
J_S<.d> [blink]
JEQ_S [blink]
JNE_S [blink]
```

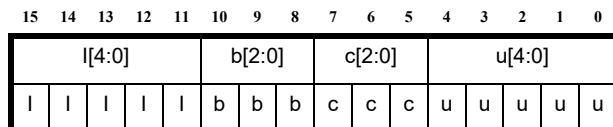
**Table 7-7 16-Bit Zero Operand Instructions**

Sub-opcode3 <b>b</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

## 7.9 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 7-15 Compact Load/Store with Offset Format**



Syntax:

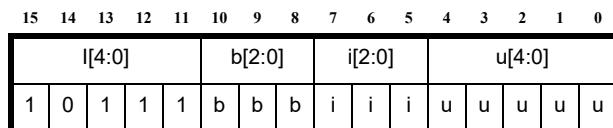
**LD\_S**     $c, [b, u7]$  ( $u7$  must be 32-bit aligned)  
**LDB\_S**     $c, [b, u5]$   
**LDH\_S**     $c, [b, u6]$  ( $u6$  must be 16-bit aligned)  
**LDH\_S.X**     $c, [b, u6]$  ( $u6$  must be 16-bit aligned)  
**ST\_S**     $c, [b, u7]$  ( $u7$  must be 32-bit aligned)  
**STB\_S**     $c, [b, u5]$   
**STH\_S**     $c, [b, u6]$  ( $u6$  must be 16-bit aligned)

**Table 7-8 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

## 7.10 Shift/Subtract/Bit Immediate [F16\_SH\_SUB\_BIT]

Figure 7-16 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 7-9 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 left
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

## 7.11 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 7-10](#). 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 7-10 Opcodes for Stack-based Operations**

Encoded Instructions		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		i[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	
LDB_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	0	1	u	u	u	u	
ST_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	1	0	u	u	u	u	
STB_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	1	1	u	u	u	u	
ADD_S	b, SP, u7	1	1	0	0	0	b	b	b	1	0	0	u	u	u	u	
ADD_S	SP, SP, u7	1	1	0	0	0	0	0	0	1	0	1	u	u	u	u	
SUB_S	SP, SP, u7	1	1	0	0	0	0	0	1	1	0	1	u	u	u	u	
POP_S	b	1	1	0	0	0	b	b	b	1	1	0	0	0	0	1	
POP_S	BLINK	1	1	0	0	0	R	R	R	1	1	0	1	0	0	1	
PUSH_S	b	1	1	0	0	0	b	b	b	1	1	1	0	0	0	1	
PUSH_S	BLINK	1	1	0	0	0	R	R	R	1	1	1	1	0	0	1	
LEAVE_S	u7	1	1	0	0	0	U	U	U	1	1	0	u	u	u	0	
ENTER_S	u6	1	1	0	0	0	0	U	U	1	1	1	u	u	u	0	

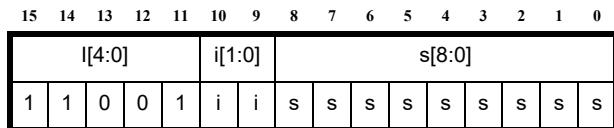
The ENTER\_S and LEAVE\_S instructions are available only when the CODE\_DENSITY extension pack is configured in the architecture. If an ENTER\_S or LEAVE\_S instruction is executed on a build without the CODE\_DENSITY extension pack, an Illegal Instruction Error exception is raised.

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.

## 7.12 Load/Add GP-Relative [F16\_GP\_LD\_ADD]

**Figure 7-17 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)*

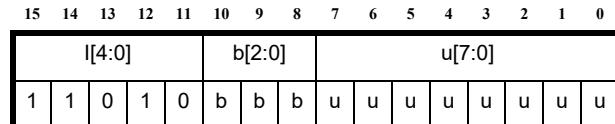
**Table 7-11 16-Bit GP Relative Instructions**

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	LD_S	r0 $\leftarrow$ mem[GP + s11].l	Load GP-relative (32-bit aligned) to r0
0x01	LDB_S	r0 $\leftarrow$ mem[GP + s9].b	Load unsigned byte GP-relative (8-bit aligned) to r0
0x02	LDH_S	r0 $\leftarrow$ 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

## 7.13 Load PCL-Relative [F16\_PCL\_LD]

The offset (u[7:0]) is shifted accordingly to provide the appropriate 32-bit data size alignment.

**Figure 7-18 Compact Load PCL Relative Format**

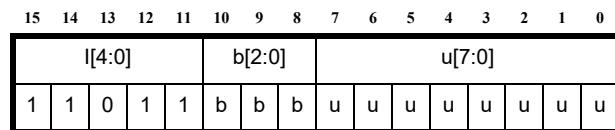


Syntax:

LD\_S b, [PCL, u10] *(32-bit aligned offset)*

## 7.14 Move Immediate [F16\_MV\_IMM]

Figure 7-19 Compact Move Immediate Format

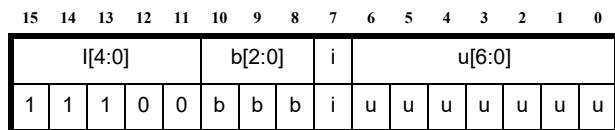


Syntax:

MOV\_S b, u8

## 7.15 ADD/CMP Immediate [F16\_OP\_IMM]

**Figure 7-20 Compact ADD/CMP Immediate Format**



Syntax:

ADD\_S b, b, u7

CMP\_S b, u7

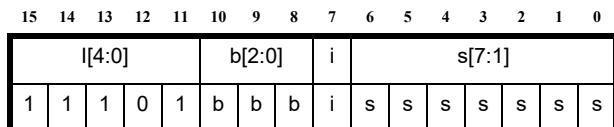
**Table 7-12 16-Bit ADD/CMP Immediate**

Sub-opcode i field (1 bit)	Instruction	Operation	Description
0x00	ADD_S	$b \leftarrow b + u7$	Add
0x01	CMP_S	$b - u7$	Compare

## 7.16 Branch on Compare Register with Zero [F16\_BCC\_REG]

The target address is 16-bit aligned.

**Figure 7-21 Compact Branch on Compare Register with Zero Format**



Syntax:

BREQ\_S b, 0, s8

BRNE\_S b, 0, s8

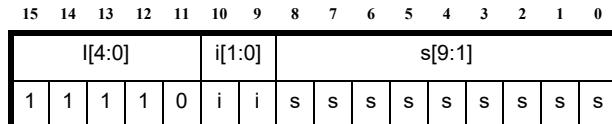
**Table 7-13 16-Bit Branch on Compare**

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

## 7.17 Branch Conditionally [F16\_BCC]

The target address is 16-bit aligned.

**Figure 7-22 Compact Branch Conditionally Format**



Syntax:

```
B_S      s10
BEQ_S   s10
BNE_S   s10
```

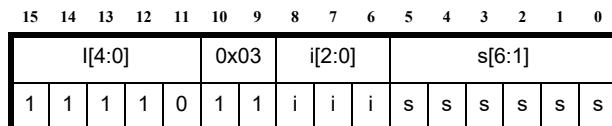
**Table 7-14 16-Bit Branch, Branch Conditionally**

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 <a href="#">Bcc_S</a>

### 7.17.1 Branch Conditionally with cc Field, 0x1E, [0x03, 0x00 – 0x07]

The target address is 16-bit aligned.

**Figure 7-23 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 7-15 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

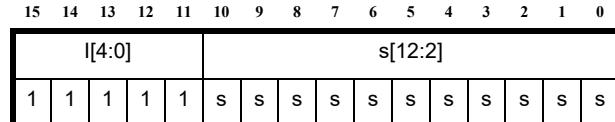


Note For details on the codes used for condition code tests, see [Table 5-7](#).

## 7.18 Branch and Link Unconditionally [F16\_BL]

The target address can only target 32-bit aligned instructions.

**Figure 7-24 Compact Branch and Link Unconditionally Format**



## Syntax:

BL\_S s13



## 8

# 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](#)” on page [251](#).

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 5-16](#) on page [265](#) lists the syntax conventions. [Table 5-5](#) on page [258](#) and [Table 5-6](#) on page [259](#) lists the encoding notation.

[Table 8-1](#) 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:

```
ADD.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](#).

## 8.1 Instruction List

**Table 8-1 List of Instructions**

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
<a href="#">ABS</a>	Absolute value	<a href="#">ABS_S</a>	Absolute value
<a href="#">ADC</a>	Add with carry		
<a href="#">ADD</a>	Add	<a href="#">ADD_S</a>	Add
<a href="#">ADD1</a>	Add with left shift by 1 bit	<a href="#">ADD1_S</a>	Add with left shift by 1 bits
<a href="#">ADD2</a>	Add with left shift by 2 bits	<a href="#">ADD2_S (see ADD2)</a>	Add with left shift by 2 bits
<a href="#">ADD3</a>	Add with left shift by 3 bits	<a href="#">ADD3_S (see ADD3)</a>	Add with left shift by 3 bits
<a href="#">AEX</a>	Swap contents of auxiliary register with a 32-bit core register		
<a href="#">AND</a>	Logical AND	<a href="#">AND_S (see AND)</a>	Logical AND
<a href="#">ASL</a>	Arithmetic Shift Left	<a href="#">ASL_S (see ASL)</a>	Arithmetic Shift Left
<a href="#">ASR</a>	Arithmetic Shift Right	<a href="#">ASR_S (see ASR)</a>	Arithmetic Shift Right
<a href="#">ASR16</a>	Arithmetic Shift Right by 16		
<a href="#">ASR8</a>	Arithmetic Shift Right by 8		
<a href="#">B</a>	Branch unconditionally	<a href="#">B_S</a>	Branch unconditionally
<a href="#">BBIT0</a>	Branch if bit equal to 0		
<a href="#">BBIT1</a>	Branch if bit equal to 1		
<a href="#">Bcc</a>	Branch if condition true	<a href="#">Bcc_S</a>	Branch if condition true
<a href="#">BCLR</a>	Clear specified bit (to 0)	<a href="#">BCLR_S (see BCLR)</a>	Clear specified bit (to 0)
<a href="#">BI</a>	Branch Indexed, 32-bit full-word table		
<a href="#">BIH</a>	Branch Indexed, 16-bit Half-word table		
<a href="#">BIC</a>	Bit-wise inverted AND	<a href="#">BIC_S (see BIC)</a>	Bit-wise inverted AND
<a href="#">BLcc</a>	Branch and Link	<a href="#">BL_S (see BLcc)</a>	Branch and Link

**Table 8-1 List of Instructions (Continued)**

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
BMSK	Bit Mask	BMSK_S (see <a href="#">BMSK</a> )	Bit Mask
BMSKN	Bit Mask Negated		
BRcc	Branch on compare	BRcc_S (see <a href="#">BRcc</a> )	Branch on compare
BRK	Break (halt) processor	BRK_S (see <a href="#">BRK</a> )	Break (halt) processor
BSET	Set specified bit (to 1)	BSET_S (see <a href="#">BSET</a> )	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 <a href="#">BTST</a> )	Test value of specified bit
BXOR	Bit XOR		
CLRI	Clear Interrupt Enable		
CMP	Compare	CMP_S (see <a href="#">CMP</a> )	Compare
DBNZ	Signed integer Divide		
DIVU	Unsigned integer Divide		
DMACH	Dual 16x16 multiply and accumulate		
DMACHU	Dual unsigned 16x16 multiply and accumulate		
DMB	Data memory barrier instruction		
DMPYH	Dual 16x16 multiplication		
DMPYHU	Dual unsigned 16x16 multiplication		
DSYNC	Synchronize instruction		
		EI_S	Execute Indexed
		ENTER_S	Function Prologue Sequence
EX	Atomic Exchange		
EXTB	Zero-extend byte to word	EXTB_S (see <a href="#">EXTB</a> )	Zero-extend byte

**Table 8-1 List of Instructions (Continued)**

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
<a href="#">EXTH</a>	Zero-extend 16-bit half-word to word	<a href="#">EXTH_S</a> (see <a href="#">EXTH</a> )	Zero-extend 16-bit half-word
<a href="#">FCVT32</a>	Conversion between 32-bit data formats		
<a href="#">FFS</a>	Find First Set		
<a href="#">FLAG</a>	Write to Status Register		
<a href="#">FLS</a>	Find Last Set		
<a href="#">FSABS</a>	Single-precision floating-point absolute operation		
<a href="#">FSADD</a>	Single-precision floating-point addition		
<a href="#">FSCMP</a>	Single-precision floating-point comparison		
<a href="#">FSCMPF</a>	Single-precision floating-point comparison		
<a href="#">FSDIV</a>	Single-precision floating-point division		
<a href="#">FSMADD</a>	Single-precision floating-point fusedMultiplyAdd A fusedMultiplyAdd is a floating-point multiply-add operation performed in one step, with a single rounding.		
<a href="#">FSMSUB</a>	Single-precision floating-point fusedMultiplySubtract A fusedMultiplySubtract is a floating-point multiply-subtract operation performed in one step, with a single rounding.		
<a href="#">FSMUL</a>	Single-precision floating-point multiplication		
<a href="#">FSNEG</a>	Single-precision floating-point negation		
<a href="#">FSSQRT</a>	Single-precision floating-point square-root		

**Table 8-1 List of Instructions (Continued)**

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
FSSUB	Single-precision floating-point subtraction		
Jcc	Jump	J_S (see <a href="#">Jcc</a> )	Jump
JL	Jump	JL_S (see <a href="#">JL</a> )	Jump and Link
		JLI_S	Jump and Link Indexed
KFLAG	Write to Status Register in kernel mode		
LD LDH LDW LDB LDD LDL	Load from memory	LD_S (see <a href="#">LD LDH</a> <a href="#">LDW LDB LDD LDL</a> )	Load from memory
LDI(see <a href="#">LDI</a> , <a href="#">LDI_S</a> )	Load Indexed	LDI_S (see <a href="#">LDI</a> , <a href="#">LDI_S</a> )	Load Indexed
		LEAVE_S	Function Epilogue Sequence
LLOCK	Load locked		
LLOCKD	Load locked on 64-bit data (ARC-32 ABI)		
LPcc	Loop (zero-overhead loops)		
LR	Load from Auxiliary memory		
LSL16	Logical Shift Left 16		
LSL8	Logical Shift Left 8		
LSR	Logical Shift Left	LSR_S (see <a href="#">LSR</a> )	Logical Shift Right
LSR16	Logical Shift Right 16		
LSR8	Logical Shift Right 8		
MAC	32x32 multiply and accumulate		
MACDU	Unsigned 32x32 multiply and accumulate, double result		
MACU	Unsigned 32x32 multiply and accumulate		
MAX	Return Maximum		
MIN	Return Minimum		
MOV	Move (copy) to register	MOV_S (see <a href="#">MOV</a> )	Move (copy) to register

**Table 8-1 List of Instructions (Continued)**

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
MPY, MPY_S	32 x 32 Signed Multiply (lsw)	MPY_S (see <a href="#">MPY, MPY_S</a> )	32 x 32 Signed Multiply (lsw)
MPYM MPYH	32 x 32 Signed Multiply (msw)		
MPYD	32x32 multiplication, double result		
MPYDU	Unsigned 32x32 multiplication, double result		
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 <a href="#">MPYUW, MPYUW_S</a> )	
MPYW, MPYW_S	16 X 16 signed multiply	MPYW_S (see <a href="#">MPYW, MPYW_S</a> )	16 x16 signed multiply
NEG	Negate	NEG_S (see <a href="#">NEG</a> )	Negate
NOP	No operation	NOP_S (see <a href="#">NOP</a> )	No operation
NORM	Normalize to 32 bits		
NORMH NORMW	Normalize to 16 bits		
NOT	Logical bit inversion	NOT_S (see <a href="#">NOT</a> )	Logical bit inversion
OR	Logical OR	OR_S (see <a href="#">OR</a> )	Logical OR
		POP_S	Restore register from stack
PREALLOC	Allocate cache line as a preparation for a store		
PREFETCH	Prefetch from memory		
PREFETCHW	Prefetch line from Memory with intention to write		
		PUSH_S	Store register to the stack
RCMP	Reverse Compare		
REM	Signed Integer Remainder		
REMU	Unsigned Integer Remainder		

**Table 8-1 List of Instructions (Continued)**

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
RLC	Rotate Left through Carry		
ROR	Rotate Right		
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		
SCOND	Store conditionally		
SCONDD	Store conditionally on 64-bit data (ARC32 ABI)		
SETcc	Set conditional		
SETI	Set Interrupt Enable		
SEXB	Sign-extend byte to word	SEXB_S (see <a href="#">SEXB</a> )	Sign-extend byte
SEXH SEXW	Sign-extend half-word to word	SEXH_S (see <a href="#">SEXH</a> <a href="#">SEXW</a> )	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 STD	Store to memory	ST_S (see <a href="#">ST</a> <a href="#">STH</a> <a href="#">STB</a> <a href="#">STD</a> )	Store to memory
SUB	Subtract	SUB_S (see <a href="#">SUB</a> )	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		

**Table 8-1 List of Instructions (Continued)**

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
<a href="#">SWAPE</a>	Swap byte ordering		
<a href="#">SWI</a>	Software interrupt	<a href="#">SWI_S</a> (see <a href="#">SWI</a> )	Software interrupt
<a href="#">SYNC</a>	Synchronize		
		<a href="#">TRAP_S</a>	Trap to system call
<a href="#">TST</a>	Test	<a href="#">TST_S</a> (see <a href="#">TST</a> )	Test
<a href="#">VADD2H</a>	Dual 16-bit vector addition		
<a href="#">VADDSUB2H</a>	Dual 16-bit vector add and subtract		
<a href="#">VMAC2H</a>	Dual 16x16 vector multiplication and accumulation		
<a href="#">VMPY2H</a>	Dual 16x16 vector multiplication		
<a href="#">VMPY2HU</a>	Dual unsigned 16x16 vector multiplication		
<a href="#">VSUB2H</a>	Two way vector subtract 16 bits		
<a href="#">VSUBADD2H</a>	Dual 16-bit vector subtract and add		
<a href="#">WEVT</a>	Enter sleep state and wait on event		
<a href="#">WLFC</a>	Enter Sleep State to reduce dynamic power during busy-waiting loops		
<a href="#">XBFU</a>	Extract unsigned bit-field		
<a href="#">XOR</a>	Logical Exclusive-OR	<a href="#">XOR_S</a> (see <a href="#">XOR</a> )	Logical Exclusive-OR
		<a href="#">UNIMP_S</a>	Unimplemented Instruction

## 8.2 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.

<b>Function</b>	Summarizes the function of the instruction.
<b>Extension Group</b>	Indicates if instruction is in the baseline set or included only as part of an extension group
<b>Operation</b>	C style expression that describes the operation of the instruction. Where relevant, a block diagram of the computation may be shown.
<b>Instruction Format</b>	Summary of the instruction format.
<b>Syntax Example</b>	Single syntax example.
<b>Flag Affected</b>	List of status flags that are affected.
<b>Description</b>	Full description of the instruction.
<b>Pseudo Code</b>	Operation of the instruction described in C style pseudo code.
<b>Assembly Code Example</b>	Assembly coding example.
<b>Syntax and Encoding</b>	The syntax of the instruction and corresponding instruction encoding. <a href="#">Table 5-16</a> on page 265 lists the instruction syntax convention. <a href="#">Table 5-5</a> on page 258 and <a href="#">Table 5-6</a> on page 259 lists the key for encoding conventions.

## ABS

### Function

Absolute

### Extension Group

BASELINE

### Operation

$b = ABS(c);$

### Instruction Format

op b, c

### Syntax Example

ABS<.f> b,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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

ABS<.f>	b, c	00100bbb00101111FBBBCCCCCC001001
ABS<.f>	b, u6	00100bbb01101111FBBBuuuuuuu001001
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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
ADC<.f>	a,b,c	00100bbb00000001FBBBBCCCCCAAAAAAA
ADC<.f>	a,b,u6	00100bbb01000001FBBBuuuuuuAAAAAAA
ADC<.f>	b,b,s12	00100bbb10000001FBBBssssssSSSSSS
ADC<.cc><.f>	b,b,c	00100bbb11000001FBBBCCCCCC0QQQQQ
ADC<.cc><.f>	b,b,u6	00100bbb11000001FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

$\text{if } (\text{cc}) \text{ a} = \text{b} + \text{c}$

### 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.



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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
ADD<.f>	a,b,c	00100bbb00000000FBBBBCCCCCAAAAAAA
ADD<.f>	a,b,u6	00100bbb01000000FBBBuuuuuuuAAAAAA
ADD<.f>	b,b,s12	00100bbb10000000FBBBssssssSSSSSS
ADD<.cc><.f>	b,b,c	00100bbb11000000FBBBCCCCC0QQQQQ
ADD<.cc><.f>	b,b,u6	00100bbb11000000FBBBuuuuuuu1QQQQQ
ADD_S	a,b,c	01100bbbccc11aaa
ADD_S	b,b,h	01110bbbhh000HH
ADD_S	h,h,s3	01110ssshhh001HH
ADD_S	b,b,limm	01110bbb11000011
ADD_S	0,limm,s3	01110sss11000111
ADD_S	b,sp,u7	11000bbb100uuuuuu
ADD_S	b,b,u7	11100bbb0uuuuuuuu
ADD_S	c,b,u3	01101bbbccc00uuu
ADD_S	SP,SP,u7	11000000101uuuuuu
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 << 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
ADD1<.f>	a,b,c	00100bbb00010100FBBBCCCCCCCCAAAAAA
ADD1<.f>	a,b,u6	00100bbb01010100FBBBuuuuuuAAAAAA
ADD1<.f>	b,b,s12	00100bbb10010100FBBBssssssssssss
ADD1<.cc><.f>	b,b,c	00100bbb11010100FBBBCCCCCCC0QQQQQ
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:

```
ADD.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 << 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
ADD2<.f>	a,b,c	00100bbb00010101FBBBCCCCCCCCAAAAAA
ADD2<.f>	a,b,u6	00100bbb01010101FBBBuuuuuuAAAAAA
ADD2<.f>	b,b,s12	00100bbb10010101FBBBssssssssssss
ADD2<.cc><.f>	b,b,c	00100bbb11010101FBBBCCCCCCC0QQQQQ
ADD2<.cc><.f>	b,b,u6	00100bbb11010101FBBBuuuuuu1QQQQQ
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:

```
ADD.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 << 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
ADD3<.f>	a,b,c	00100bbb00010110FBBBCCCCCCCCAAAAAA
ADD3<.f>	a,b,u6	00100bbb01010110FBBBuuuuuuAAAAAA
ADD3<.f>	b,b,s12	00100bbb10010110FBBBssssssssssss
ADD3<.cc><.f>	b,b,c	00100bbb11010110FBBBCCCCCCC0QQQQQ
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:

```
ADD.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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
AEX	b, [c]	00100 <b>bbb</b> 00100111RBBBCCCCCRRRRR
AEX	b, [u6]	00100 <b>bbb</b> 01100111RBBBuuuuuuRRRRR
AEX	b, [s12]	00100 <b>bbb</b> 10100111RBBBssssssSSSSS
AEX<.cc>	b, [c]	00100 <b>bbb</b> 11100111RBBBCCCCC0QQQQQ
AEX<.cc>	b, [u6]	00100 <b>bbb</b> 11100111RBBBuuuuuu1QQQQQ

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
AND<.f>	a, b, c	00100 <b>bbb</b> 00000100 <b>FBBB</b> CCCCCCAAAAAA
AND<.f>	a, b, u6	00100 <b>bbb</b> 01000100 <b>FBBB</b> uuuuuuAAAAAA
AND<.f>	b, b, s12	00100 <b>bbb</b> 10000100 <b>FBBB</b> ssssssSSSSSS
AND<.cc><.f>	b, b, c	00100 <b>bbb</b> 11000100 <b>FBBB</b> CCCCCC0QQQQQ
AND<.cc><.f>	b, b, u6	00100 <b>bbb</b> 11000100 <b>FBBB</b> uuuuuu1QQQQQ
AND_S	b, b, c	01111 <b>bbbccc</b> 00100

## 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:

```
ADD.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

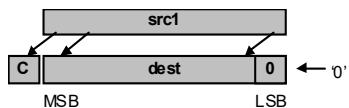
Arithmetic Shift Left

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

ASL<.f>	b, c	00100bbb00101111FBBBCCCCC000000
ASL<.f>	b, u6	00100bbb01101111FBBBuuuuuuu000000
ASL_S	b, c	01111bbbccc11011

## ASL Multiple

### Function

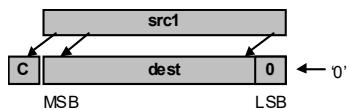
Multiple Arithmetic Shift Left

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
ASL<.f>	a,b,c	00101bbb00000000FBBBCCCCCAAAAAA
ASL<.f>	a,b,u6	00101bbb01000000FBBBuuuuuuAAAAAA
ASL<.f>	b,b,s12	00101bbb10000000FBBBssssssSSSSSS
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:

```
ADD.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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

ASR<.f>	b, c	00100bbb00101111FBBBCCCCC000001
ASR<.f>	b, u6	00100bbb01101111FBBBuuuuuuu000001
ASR_S	b, c	01111bbbccc11100

## ASR multiple

### Function

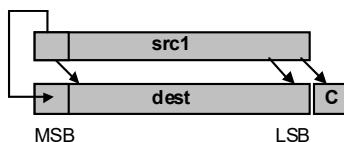
Multiple Arithmetic Shift Right

### 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	<input checked="" type="checkbox"/>	= Set if result is zero
N	<input checked="" type="checkbox"/>	= Set if most-significant bit of result is set
C	<input checked="" type="checkbox"/>	= Set if carry is generated
V	<input type="checkbox"/>	= 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)
  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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
ASR<.f>	a,b,c	00101bbb00000010FBBBCCCCCCCCAAAAAA
ASR<.f>	a,b,u6	00101bbb01000010FBBBuuuuuuAAAAAA
ASR<.f>	b,b,s12	00101bbb10000010FBBBssssssSSSSSS
ASR<.cc><.f>	b,b,c	00101bbb11000010FBBBCCCCCCC0QQQQQ
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:

```
ADD.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](#).

## ASRL Multiple

### Function

Multiple Arithmetic Shift Right Long

### Extension Group

ARC64 ISA

### Operation

if (cc) a = (signed) b >> c;



### Instruction Format

op a, b, c

### Syntax Example

ASRL<.f> a,b,c

### STATUS32 Flags Affected

Z	<input checked="" type="checkbox"/>	= Set if result is zero
N	<input checked="" type="checkbox"/>	= Set if most-significant bit of result is set
C	<input checked="" type="checkbox"/>	= Set if carry is generated
V	<input type="checkbox"/>	= Unchanged

### Description

Arithmetically, shift 64-bit operand b right by c places and place the result in the 64-bit destination register a. Only the bottom 6 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                                /* ASRL */
  dest = ((signed)src1) >> (src2 & 63)
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[63]
    C_flag = if src2==0 then 0 else src1[src2-1]
  
```

## Assembly Code Example

```

ASRL r1,r2,r3      ; Arithmetic shift right contents of r2 by r3 bits
                     ; and write result into r1
  
```

## Syntax and Encoding

Instruction Code		
ASRL<.f>	a,b,c	01011bbb00100010FBBBCCCCCCCCAAAAAA
ASRL<.f>	a,b,u6	01011bbb01100010FBBBuuuuuuAAAAAA
ASRL<.f>	b,b,s12	01011bbb10100010FBBBssssssSSSSSS
ASRL<.cc><.f>	b,b,c	01011bbb11100010FBBBCCCCCCC0QQQQQ
ASRL<.cc><.f>	b,b,u6	01011bbb11100010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

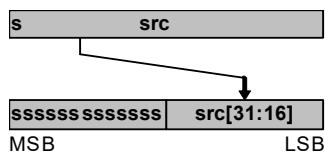
Arithmetic Shift Right 16

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

ASR16<.f>	b,c	00101bbb00101111FBBBCCCCC001100
ASR16<.f>	b,u6	00101bbb01101111FBBBuuuuuu001100

## ASR8

### Function

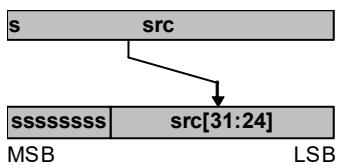
Arithmetic Shift Right 8

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

ASR8<.f>	b,c	00101 <b>bbb</b> 00101111 <b>FBBB</b> CCCCCC <b>001101</b>
ASR8<.f>	b,u6	00101 <b>bbb</b> 01101111 <b>FBBB</b> uuuuuu <b>001101</b>

**B****Function**

Branch unconditionally

**Extension Group**

BASELINE

**Operation**

$\text{PC} = (\text{PCL} + \text{s25});$

**Instruction Format**

op s25

**Syntax Example**

$\text{B} <.\text{d}> \text{s25}$

**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 5-37](#) on page [285](#) describes the delay slot modes, .d.

[Table 5-7](#) on page [260](#) 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

B<.d>      s25    0000ssssssssss1SSSSSSSSN Rttt

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

```
B_S      s10    1111000ssssssss
```

## BBIT0

### Function

Branch on Bit Test Clear

### Extension Group

BASELINE

### Operation

if ((b AND (1<<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 5-37 on page 285 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.

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 5-9](#) on page [262](#), and more details can be found in the section “[Branch On Compare or Bit Test](#)” on page [287](#). [Table 5-9](#) on page [262](#) lists the interpretation of the static branch prediction syntax <.T>.

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

BBIT0<.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCCN0Y110
BBIT0<.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuuuN1Y110

## 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 5-37](#) on page [285](#) 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.

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](#)” on page [312](#), and more details can be found in section “[Branch On Compare or Bit Test](#)” on page [287](#). Table [5-9](#) on page [262](#) lists the interpretation of the static branch prediction syntax <.T>.

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### 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 5-37](#) on page [285](#).

The condition codes, cc, are described in [Table 5-7](#) on page [260](#).



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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

B<cc><.d> s21 00000ssssssss0SSSSSSSSNQQQQ

## 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 5-7 on page 260 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  
BPL_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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

BEQ_S	s10	1111001sssssssss
BNE_S	s10	1111010sssssssss
BGT_S	s7	1111011000ssssss
BGE_S	s7	1111011001ssssss
BLT_S	s7	1111011010ssssss
BLE_S	s7	1111011011ssssss
BHI_S	s7	1111011100ssssss
BHS_S	s7	1111011101ssssss
BLO_S	s7	1111011110ssssss
BLS_S	s7	1111011111ssssss

## BCLR

### Function

Bit Clear

### Extension Group

BASELINE

### Operation

`if (cc) then a = (b & ~(1<< 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
BCLR<.f>	a,b,c	00100bbb00010000FBBBCCCCCAAAAAA
BCLR<.f>	a,b,u6	00100bbb01010000FBBBuuuuuuAAAAAA
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:

```
ADD.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**

$PC = \text{next\_PC} + (c \ll 2)$

**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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

BI [c] 00100RRR001001000RRRCCCCCRRRRR

## BIC

### Function

Bitwise AND Operation with Inverted Source

### Extension Group

BASELINE

### Operation

`if (cc) a = b & ~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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
BIC<.f>	a, b, c	00100bbb00000110FBBCCCCCCAAAAAA
BIC<.f>	a, b, u6	00100bbb01000110FBBCBuuuuuuAAAAAA
BIC<.f>	b, b, s12	00100bbb10000110FBBCBssssssSSSSSS
BIC<.cc><.f>	b, b, c	00100bbb11000110FBBCCCCCC0QQQQQ
BIC<.cc><.f>	b, b, u6	00100bbb11000110FBBCBuuuuuu1QQQQQ
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:

```
ADD.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

$$\text{PC} = \text{next\_PC} + (\text{c} \ll 1)$$

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

BIH [c] 00100RRR001001010RRRCCCCCRRRRR

## 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 5-37](#) on page [285](#).

The condition codes, cc, are described in [Table 5-7](#) on page [260](#).



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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

```

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	00001 <b>ssssssss</b> 00 <b>ssssssss</b> N <b>QQQQ</b>
BL<.d>	s25	00001 <b>ssssssss</b> 10 <b>ssssssss</b> NR <b>tttt</b>
BL_S	s13	11111 <b>ssssssssss</b>

## BMSK

### Function

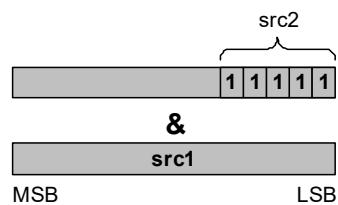
Bit Mask

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
BMSK<.f>	a,b,c	00100bbb00010011FBBBCCCCCCCCAAAAAA
BMSK<.f>	a,b,u6	00100bbb01010011FBBBuuuuuuAAAAAA
BMSK<.f>	b,b,s12	00100bbb10010011FBBBssssssssssss
BMSK<.cc><.f>	b,b,c	00100bbb11010011FBBBCCCCCCC0QQQQQ
BMSK<.cc><.f>	b,b,u6	00100bbb11010011FBBBuuuuuu1QQQQQ
BMSK_S	b,b,u5	10111bbb110uuuuu

## 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:

```
ADD.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

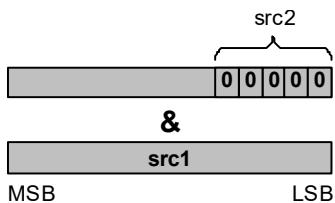
Bit Mask Negated

### Extension Group

BASELINE

### Operation

if (cc) then a = (b & ( $\sim 0 \ll (c+1)$ ));



### 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 & 0xFFFFFE00)
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
BMSKN<.f>	a,b,c	00100 <b>bbb00101100FBBBCCCCCCCCAAAAAA</b>
BMSKN<.f>	a,b,u6	00100 <b>bbb01101100FBBBuuuuuuAAAAAA</b>
BMSKN<.f>	b,b,s12	00100 <b>bbb10101100FBBBssssssSSSSSS</b>
BMSKN<.cc><.f>	b,b,c	00100 <b>bbb11101100FBBBCCCCCCC0QQQQQ</b>
BMSKN<.cc><.f>	b,b,u6	00100 <b>bbb11101100FBBBuuuuuu1QQQQQ</b>

## 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:

```
ADD.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 5-37](#) on page [285](#), 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](#)” on page [312](#), and more details can be found in section “[Branch On Compare or Bit Test](#)” on page [287](#). The interpretation of the <.T> syntax can be found in [Table 5-9](#) on page [262](#).

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
BREQ <.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCN0Y000
BREQ <.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuuN1Y000
BRNE <.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCN0Y001
BRNE <.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuuN1Y001
BRLT <.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCN0Y010
BRLT <.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuuN1Y010
BRGE <.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCN0Y011
BRGE <.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuuN1Y011

BRLO <.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCN0Y100
BRLO <.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuuN1Y100
BRHS <.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCN0Y101
BRHS <.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuuN1Y101
BRNE_S	b, 0, s8	11101bbb1sssssss
BREQ_S	b, 0, s8	11101bbb0sssssss

## 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	<input type="checkbox"/>	= Unchanged
N	<input type="checkbox"/>	= Unchanged
C	<input type="checkbox"/>	= Unchanged
V	<input type="checkbox"/>	= Unchanged
BH	<input checked="" type="checkbox"/>	= 1
H	<input checked="" type="checkbox"/>	= 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.



**Note** 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
DEBUG[H] = 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

BRK        0010010101101110000RRRRRR111111

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
  dest = src1 OR (1 << (src2 & 31))          /* BSET */
  if F==1 then
    Z_flag = if dest==0 then 1 else 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
BSET<.f>	a,b,c	00100bbb00001111FBBBCCCCC <del>AAAAAA</del>
BSET<.f>	a,b,u6	00100bbb01001111FBBBuuuuuu <del>AAAAAA</del>
BSET<.f>	b,b,s12	00100bbb10001111FBBBssssss <del>SSSSSS</del>
BSET<.cc><.f>	b,b,c	00100bbb11001111FBBBCCCCC <del>0QQQQQ</del>
BSET<.cc><.f>	b,b,u6	00100bbb11001111FBBBuuuuuu <del>1QQQQQ</del>
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:

```
ADD.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
```

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update.

### 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	00101bbb00101111FBBBBCCCCC101110
BSPEEK<.F>	b, u6	00101bbb01101111FBBBBuuuuuu101110

## 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



**Note** You cannot use the XY operands as a source or destination with this instruction.

### Syntax Example

BSPOP<.F> b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update

### 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	00101bbb00101111FBBBCCCCC101111
BSPOP<.F>	b,u6	00101bbb01101111FBBBuuuuuu101111

## 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



**Note** You cannot use the XY operands as a source or destination with this instruction.

### Syntax Example

BSPUSH a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update

### 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
/* BSPUSH */

// 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 << nbts) - 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 - nbts - bit_os);
    v = (v & ~m) | (val64 << (64 - nbts - bit_os));
    bit_os += nbts
    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 += nbts
    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 += nbts
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:

```
ADD.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.



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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
BTST	b,c	00100 <b>bbb</b> 000100011 <b>BBB</b> CCCCCC <b>RRRRRR</b>
BTST	b,u6	00100 <b>bbb</b> 010100011 <b>BBB</b> uuuuuu <b>RRRRRR</b>
BTST	b,s12	00100 <b>bbb</b> 100100011 <b>BBB</b> ssssss <b>SSSSSS</b>
BTST<.cc>	b,c	00100 <b>bbb</b> 110100011 <b>BBB</b> CCCCCC <b>0QQQQQ</b>
BTST<.cc>	b,u6	00100 <b>bbb</b> 110100011 <b>BBB</b> uuuuuu <b>1QQQQQ</b>
BTST_S	b,u5	10111 <b>bbb</b> 111uuuuu

## 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:

```
ADD.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 ^ (1 << 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
BXOR<.f>	a,b,c	00100bbb00010010FBBBCCCCC <del>AAAAAA</del>
BXOR<.f>	a,b,u6	00100bbb01010010FBBBuuuuuu <del>AAAAAA</del>
BXOR<.f>	b,b,s12	00100bbb10010010FBBBssssss <del>SSSSSS</del>
BXOR<.cc><.f>	b,b,c	00100bbb11010010FBBBCCCCC <del>0QQQQQ</del>
BXOR<.cc><.f>	b,b,u6	00100bbb11010010FBBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

{dest  $\leftarrow$  { 26'd0, 1'b1, STATUS32.IE, STATUS32.E[3:0] }; 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 Only Access](#) exception.

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

```
{
    dst <- { 26'd0, 1'b1, STATUS32.IE, STATUS32.E[3:0] }           /* CLRI */
    STATUS32.IE <- 0
}

if (SEC_MODES_OPTION == 1)/*SecureShield is present*/                  /* CLRI * if
if ((STATUS32.U == 0) && ((STATUS32.S == 1) || (SEC_STAT.NIC == 1)))   SecureShield
dst = (0 | (1 << 6) | (STATUS32.IE << 5) | (STATUS32.E))          component is
STATUS32.IE = 0                                                       configured.

else /* SecureShield is not present */
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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### 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.



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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DADDH11*/
```

## 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	00110bbb00110100FBBBCCCCC <del>AAAAAA</del>
DADDH11 <.f>	a,b,limm	00110bbb00110100FBBB111110 <del>AAAAAA</del>
DADDH11 <.f>	a,limm,c	0011011000110100F111CCCCC <del>AAAAAA</del>

#### Without Result

DADDH11<.f>	0,b,c	00110bbb00110100FBBBCCCCC111110
DADDH11 <.f>	0,b,limm	00110bbb00110100FBBB111110111110
DADDH11 <.f>	0,limm,c	0011011000110100F111CCCCC111110

## Related Instructions

[DADDH12](#)   [DSYNC](#)

[DADDH21](#)   [DSUBH11](#)

[DADDH22](#)

## 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 writeback result is zero.		= Limm Data
N		= Set if MSB of writeback result is set.		
C		= Set if writeback result is a NaN.		
V		= Set if writeback 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

```
/*DADDH12*/
```

## 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	00110bbb00110101FBBBCCCCCAAAAAA
DADDH12 <.f>	a,b,limm	00110bbb00110101FBBB111110AAAAAA
DADDH12 <.f>	a,limm,c	0011011000110101F111CCCCCAAAAAA

#### Without Result

DADDH12 <.f>	0,b,c	00110bbb00110101FBBBCCCCC111110
DADDH12 <.f>	0,b,limm	00110bbb00110101FBBB111110111110
DADDH12 <.f>	0,limm,c	0011011000110101F111CCCCC111110

## 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DADDH21*/
```

## 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DADDH22*/
```

## 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	00110bbb00110111FBBBCCCCCCAAAAAA
DADDH22 <.f>	a,b,limm	00110bbb00110111FBBB111110AAAAAA
DADDH22 <.f>	a,limm,c	0011011000110111F111CCCCCCAAAAAA

#### Without Result

DADDH22 <.f>	0,b,c	00110bbb00110111FBBBCCCCCC111110
DADDH22 <.f>	0,b,limm	00110bbb00110111FBBB111110111110
DADDH22 <.f>	0,limm,c	0011011000110111F111CCCCCC111110

## Related Instructions

<a href="#">DBNZ</a>	<a href="#">DMULH22</a>
<a href="#">DADDH12</a>	<a href="#">DSUBH22</a>
<a href="#">DADDH21</a>	

## 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		= Unaffected		= Limm Data
N		= Unaffected		
C		= Unaffected		
V		= Unaffected		

### 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

```
/*DEXCL1*/
```

## Assembly Code Example

```
DEXCL1 r1,r2,r3      ;Register D1 Exchange
```

## Syntax and Encoding

### Instruction Code

#### With Result

DEXCL1	a,b,c	00110bbb001111000BBBCCCCC <del>AAAAAA</del>
DEXCL1	a,b,limm	00110bbb001111000BBB111110 <del>AAAAAA</del>
DEXCL1	a,limm,c	00110110001111000111CCCCCC <del>AAAAAA</del>

#### Without Result

DEXCL1	0,b,c	00110bbb001111000BBBCCCCC111110
DEXCL1	0,b,limm	00110bbb001111000BBB111110111110
DEXCL1	0,limm,c	00110110001111000111CCCCCC111110

## 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		= 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

```
/*DEXCL2*/
```

### Assembly Code Example

```
DEXCL2 r1,r2,r3      ;Register D2 Exchange
```

## Syntax and Encodings

### Instruction Code

#### With Result

DEXCL2	a,b,c	00110bbb001111010BBBCCCCC <del>AAAAAA</del>
DEXCL2	a,b,limm	00110bbb001111010BBB111110 <del>AAAAAA</del>
DEXCL2	a,limm,c	00110110001111010111CCCCCC <del>AAAAAA</del>

#### Without Result

DEXCL2	0,b,c	00110bbb001111010BBBCCCCC <del>111110</del>
DEXCL2	0,b,limm	00110bbb001111010BBB111110 <del>111110</del>
DEXCL2	0,limm,c	00110110001111010111CCCCCC <del>111110</del>

## Related Instructions

[DEXCL1](#)

## DBNZ

### Function

Decrement and Branch if Non-zero

### Extension Group

BASELINE

### Operation

`if (src != 1) PC = PCL + s13; src = src - 1`

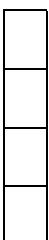
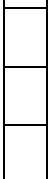
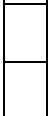
### Instruction Format

`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 will raise an Illegal Instruction exceptions. The register operand must be a writable general-purpose register, i.e. 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 will also raise 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 5-37](#) on page [285](#).



**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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

DBNZ<.d>	b, s13	00100bbb1000110N0BBBssssssssss
----------	--------	--------------------------------

# DIV

## Function

2's Complement Integer Divide.

## Extension Group

DIV\_Rem\_Option

## 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, please refer to [Appendix Table C-1, "Implementation behavior on division overflow"](#).

## Pseudo Code

```

if (cc == true) {
    if ((src2 != 0) && ((src1 != 0x80000000) || src2 !=xffffffff))
    {
        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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
DIV<.f>	a,b,c	00101bbb00000100FBBCCCCCAAAAAA
DIV<.f>	a,b,u6	00101bbb01000100FBBCuuuuuuAAAAAA
DIV<.f>	b,b,s12	00101bbb10000100FBBCssssssSSSSSS
DIV<.cc><.f>	b,b,c	00101bbb11000100FBBCCCCCC0QQQQQ
DIV<.cc><.f>	b,b,u6	00101bbb11000100FBBCuuuuuu1QQQQQ

## 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:

```
ADD.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.

### Extension Group

DIV\_REM\_OPTION

### 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, please refer to [Appendix Table C-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) {                                     /* DIVU */
    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

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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

Instruction Code		
DIVU<.f>	a,b,c	00101bbb00000101FBBCCCCCAAAAAA
DIVU<.f>	a,b,u6	00101bbb01000101FBBDuuuuuuAAAAAA
DIVU<.f>	b,b,s12	00101bbb10000101FBBDssssssSSSSSS
DIVU<.cc><.f>	b,b,c	00101bbb11000101FBBDCCCCC0QQQQQ
DIVU<.cc><.f>	b,b,u6	00101bbb11000101FBBDuuuuuu1QQQQQ

## 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:

```
ADD.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

This section describes the standard DMACH instruction.

### Function

Dual 16x16 signed integer multiply and accumulate.

### Extension Group

MPY\_OPTION == 7

### Operation

```
if (cc) {
    result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
    a = result.w0;
    acc = result;
}
```

### 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 accumulator to form the result. The least-significant 32 bits of the 64-bit result are assigned to the destination register, and the 64-bit result is assigned to the 64-bit accumulator (ACCH, ACCL).

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is set if an overflow occurs during accumulation. This instruction never clears the V flag. The sign flag (N) is set to the sign of the accumulator result.

## Pseudo Code

```

if(cc) {
    result = acc + (b.h0 * c.h0) + (b.h1 * c.h1); /* DMACH */
    a = result.w0;
    acc = result;
}

```

## Assembly Code Example

DMACH r1,r2,r3	; dual 16x16 MAC of r2 and r3
----------------	-------------------------------

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
DMACH<.f>	a,b,c	00101 <b>bbb</b> 00010010F <b>BBB</b> CCCCCC <u>AAAAAA</u>
DMACH<.f>	a,b,u6	00101 <b>bbb</b> 01010010F <b>BBB</b> uuuuuu <u>AAAAAA</u>
DMACH<.f>	b,b,s12	00101 <b>bbb</b> 10010010F <b>BBB</b> ssssss <u>SSSSSS</u>
DMACH<.cc><.f>	b,b,c	00101 <b>bbb</b> 11010010F <b>BBB</b> CCCCCC <u>QQQQQQ</u>
DMACH<.cc><.f>	b,b,u6	00101 <b>bbb</b> 11010010F <b>BBB</b> uuuuuu <u>1QQQQQ</u>

## 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:

ADD.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

This section describes the standard DMACHU instruction.

### Function

Dual unsigned integer 16x16 multiply and accumulate

### Extension Groups

MPY\_OPTION > 6

### 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. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### 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 to generate two 32-bit products. The two products are added to the accumulator to form the result. The least-significant 32 bits of the 64-bit result are assigned to the destination register, and the 64-bit result is assigned to the 64-bit accumulator (ACCH, ACCL).

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is set if an overflow occurs during accumulation. This instruction never clears the V flag.

## Pseudo Code

```

if(cc) {
    result = acc + (b.h0 * c.h0) + (b.h1 * c.h1);
    a = result.w0;
    acc = result;
}
/* DMACHU */

```

## Assembly Code Example

DMACHU r1,r2,r3	; dual 16x16 unsigned MAC of r2 ;and r3
-----------------	---

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
DMACHU<.f>	a,b,c	00101bbb00010011FBBBCCCCCCCCAAAAAA
DMACHU<.f>	a,b,u6	00101bbb01010011FBBBuuuuuuAAAAAA
DMACHU<.f>	b,b,s12	00101bbb10010011FBBBssssssSSSSSS
DMACHU<.cc><.f>	b,b,c	00101bbb11010011FBBBCCCCCCCC0QQQQQ
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:

ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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

This section describes the standard DMPYH instruction.

### Function

Sum of dual 16x16 multiplication

### Extension Group

MPY\_OPTION > 6

### 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. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### 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 to generate two 32-bit results. Assign the least-significant 32 bits to the destination register, and assign the sum of the two 32-bit products to the 64-bit accumulator (ACCH, ACCL).

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is always cleared, and the sign flag (N) is set to the sign of the accumulator result.

## Pseudo Code

```

if(cc) {                                     /* DMPYH */
    result = (b.h0 * c.h0) + (b.h1 * c.h1);
    a = result.w0;
    acc = result;
}

```

## Assembly Code Example

DMPYH r1,r2,r3	; Sum of dual 16x16 multiply of r2 ;and r3
----------------	--

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
DMPYH<.f>	a,b,c	00101bbb00010000FBBBCCCCCCCCAAAAAA
DMPYH<.f>	a,b,u6	00101bbb01010000FBBBuuuuuuAAAAAA
DMPYH<.f>	b,b,s12	00101bbb10010000FBBBssssssSSSSSS
DMPYH<.cc><.f>	b,b,c	00101bbb11010000FBBBCCCCCCC0QQQQQ
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:

ADD.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

This section describes the standard DMPYHU instruction.

### Function

Sum of dual unsigned 16x16 multiplication

### Extension Group

MPY\_OPTION > 6

### 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. For more information about 64-bit operands, see “Data Formats” on page 80.

### 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 to generate two 32-bit results. Assign the least-significant 32 bits to the destination register, and assign the sum of the two 32-bit products to the 64-bit accumulator (ACCH, ACCL).

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is always cleared.

## Pseudo Code

```

if(cc) {
    result = (b.h0 * c.h0) + (b.h1 * c.h1); /* DMPYHU */
    a = result.w0 ;
    acc = result;
}

```

## Assembly Code Example

```
DMPYHU r1,r2,r3 ; sum of dual 16x16 unsigned multiply of r2 and r3
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
DMPYHU<.f>	a,b,c	00101bbb00010001FBBBCCCCC <del>AAAAAA</del>
DMPYHU<.f>	a,b,u6	00101bbb01010001FBBBuuuuuu <del>AAAAAA</del>
DMPYHU<.f>	b,b,s12	00101bbb10010001FBBBssssss <del>SSSSSS</del>
DMPYHU<.cc><.f>	b,b,c	00101bbb11010001FBBBCCCCC <del>0QQQQQ</del>
DMPYHU<.cc><.f>	b,b,u6	00101bbb11010001FBBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## DSYNC

### Function

Data transfer synchronization instruction

### Extension Group

OS\_OPT\_OPTION == 1. This option cannot be configured in the ARC EM processor, and hence this instruction is decoded as an illegal instruction in the ARC EM processor.

### Operation

During data synchronization, wait for completion of all outstanding data memory transactions before any new operations can begin

### Instruction Format

op

### Syntax Example

DSYNC

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

This instruction is similar to the SYNC instruction.

For data synchronization, the purpose of the DSYNC instruction is to ensure that all outstanding data memory operations started by the processor have finished before any new operations of any kind can begin

DSYNC requires the core to wait for all pending memory operations, including cache, BPU, and TLB maintenance operations to finish.

DSYNC does not wait for completion of outstanding non-memory operations, such as committed FPU, divider, or APEX operations that have not yet retired.

## Pseudo Code

```
do                                /*DSYNC
null
until not (load_pending or store_pending)
```

## Assembly Code Example

```
DSYNC      ; synchronize
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

DSYNC	00100010011011110001RRRRRR111111
-------	----------------------------------

## DMB

### Function

Data memory barrier

### Extension Group

OS\_OPT\_OPTION == 1. This option cannot be configured in the ARC EM processor, and hence this instruction is decoded as an illegal instruction in the ARC EM processor.

### Operation

Wait for completion of selected type of data memory operations before initiating similar types of data memory operations

### Instruction Format

op u3

### Syntax Example

DMB u3

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

The Data Memory Barrier (DMB) instruction ensures that selected types of memory-based operations issued before the DMB instruction are completed before initiation of any similar types of memory-based operations after the DMB instruction.

Following are the types of memory-based operations affected by the DMB instruction:

- ❑ **Read:** load or pop instructions including those from LEAVE\_S and interrupt or exception epilogues.
- ❑ **Write:** store or push instructions including those from ENTER\_S and interrupt or exception prologues.
- ❑ **Control:** cache or TLB maintenance operations, and all SYNC, DSYNC, and DMB instructions.

## Pseudo Code

```
LD R0, [R3]                                /*DMB
DMB
LD R1, [R2]
```

## Assembly Code Example

```
DMB          ; Data memory barrier
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

DMB	u3	00100011011011110001RRRuuu111111
-----	----	----------------------------------

The DMB u3 operand is a bit-vector that defines the subset of operation types that are included in the memory barrier.

- ❑ Bit 0: Read
- ❑ Bit 1: Write
- ❑ Bit 2: Control

## 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DMULH11*/
```

## 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	00110bbb00110000FBBBCCCCAA
DMULH11 <.f>	a,b,limm	00110bbb00110000FBBB11110A
DMULH11 <.f>	a,limm,c	0011011000110000F111CCCCAA

#### Without Result

DMULH11 <.f>	0,b,c	00110bbb00110000FBBBCCCC11
DMULH11 <.f>	0,b,limm	00110bbb00110000FBBB111101
DMULH11 <.f>	0,limm,c	0011011000110000F111CCCC11

## Related Instructions

[DMULH12](#)   [DBNZ](#)

[DMULH21](#)   [DSUBH11](#)

[DMULH22](#)

## DMULH12

Multiply by D2, writing to D1

### Extension Group

Double-precision Floating Point

### Operation

- $D1 = D2 * (\text{src1}:\text{src2})$
- dest =  $(D2 * (\text{src1}:\text{src2})).\text{high}$

### 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DMULH12*/
```

## 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	00110bbb00110001FBBBBCCCCC <del>AAAAAA</del>
DMULH12 <.f>	a,b,limm	00110bbb00110001FBBBB111110 <del>AAAAAA</del>
DMULH12 <.f>	a,limm,c	0011011000110001F111CCCCC <del>AAAAAA</del>

#### Without Result

DMULH12 <.f>	0,b,c	00110bbb00110001FBBBBCCCCC111110
DMULH12 <.f>	0,b,limm	00110bbb00110001FBBBB111110111110
DMULH12 <.f>	0,limm,c	0011011000110001F111CCCCC111110

## Related Instructions

[DSYNC](#)    [DADDH12](#)

[DMULH21](#)    [DSUBH12](#)

[DMULH22](#)

## 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DMULH21*/
```

## 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	00110bbb00110010FBBBBCCCCC <del>AAAAAA</del>
DMULH21 <.f>	a,b,limm	00110bbb00110010FBBBB11110 <del>AAAAAA</del>
DMULH21 <.f>	a,limm,c	0011011000110010F111CCCCC <del>AAAAAA</del>

#### Without Result

DMULH21 <.f>	0,b,c	00110bbb00110010FBBBBCCCCC11110
DMULH21 <.f>	0,b,limm	00110bbb00110010FBBBB11110111110
DMULH21 <.f>	0,limm,c	0011011000110010F111CCCCC11110

## Related Instructions

<a href="#">DSYNC</a>	<a href="#">DADDH21</a>
<a href="#">DMULH12</a>	<a href="#">DSUBH21</a>
<a href="#">DMULH22</a>	

## 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DMULH22*/
```

## 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	00110bbb00110011FBBBBCCCCC <del>AAAAAA</del>
DMULH22 <.f>	a,b,limm	00110bbb00110011FBBB111110 <del>AAAAAA</del>
DMULH22 <.f>	a,limm,c	0011011000110011F111CCCCC <del>AAAAAA</del>

#### Without Result

DMULH22 <.f>	0,b,c	00110bbb00110011FBBBBCCCCC111110
DMULH22 <.f>	0,b,limm	00110bbb00110011FBBB111110111110
DMULH22 <.f>	0,limm,c	0011011000110011F111CCCCC111110

## Related Instructions

<a href="#">DSYNC</a>	<a href="#">DADDH22</a>
<a href="#">DMULH12</a>	<a href="#">DSUBH22</a>
<a href="#">DMULH21</a>	

## 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DSUBH11*/
```

## 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	00110bbb00111000FBBBCCCCAA
DSUBH11 <.f>	a,b,limm	00110bbb00111000FBBB11110A
DSUBH11 <.f>	a,limm,c	0011011000111000F111CCCCAA

#### Without Result

DSUBH11 <.f>	0,b,c	00110bbb00111000FBBBCCCC11
DSUBH11 <.f>	0,b,limm	00110bbb00111000FBBB111101
DSUBH11 <.f>	0,limm,c	0011011000111000F111CCCC11

## Related Instructions

[DSUBH12](#)   [DBNZ](#)

[DSUBH21](#)   [DSYNC](#)

[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 writeback result is zero	L	= Limm Data
N	•	= Set if MSB of writeback result is set		
C	•	= Set if writeback result is a NaN		
V	•	= Set if writeback 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

```
/*DSUBH12*/
```

## 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	00110bbb00111001FBBBBCCCCC <del>AAAAAA</del>
DSUBH12 <.f>	a,b,limm	00110bbb00111001FBBBB111110 <del>AAAAAA</del>
DSUBH12 <.f>	a,limm,c	0011011000111001F111CCCCC <del>AAAAAA</del>

#### Without Result

DSUBH12 <.f>	0,b,c	00110bbb00111001FBBBBCCCCC111110
DSUBH12 <.f>	0,b,limm	00110bbb00111001FBBBB111110111110
DSUBH12 <.f>	0,limm,c	0011011000111001F111CCCCC111110

## 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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

```
/*DSUBH21*/
```

## 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	00110bbb00111010FBBBBCCCCC <del>AAAAAA</del>
DSUBH21 <.f>	a,b,limm	00110bbb00111010FBBBB111110 <del>AAAAAA</del>
DSUBH21 <.f>	a,limm,c	0011011000111010F111CCCCC <del>AAAAAA</del>

#### Without Result

DSUBH21 <.f>	0,b,c	00110bbb00111010FBBBBCCCCC <del>111110</del>
DSUBH21 <.f>	0,b,limm	00110bbb00111010FBBBB111110 <del>111110</del>
DSUBH21 <.f>	0,limm,c	0011011000111010F111CCCCC <del>111110</del>

## 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 writeback result is zero		= Limm Data
N		= Set if MSB of writeback result is set		
C		= Set if writeback result is a NaN		
V		= Set if writeback 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	00110bbb00111011FBBBCCCCCCAAAAAA
DSUBH22 <.f>	a,b,limm	00110bbb00111011FBBB111110AAAAAA
DSUBH22 <.f>	a,limm,c	0011011000111011F111CCCCCCAAAAAA

#### Without Result

DSUBH22 <.f>	0,b,c	00110bbb00111011FBBBCCCCC111110
DSUBH22 <.f>	0,b,limm	00110bbb00111011FBBB111110111110
DSUBH22 <.f>	0,limm,c	0011011000111011F111CCCCCC111110

## Related Instructions

[DSUBH11](#)    [DADDH22](#)

[DSUBH12](#)    [DMULH22](#)

[DSUBH21](#)

## EL\_S

### Function

Execute Indexed

### Extension Group

CODE\_DENSITY option 3

### Operation

Execute instruction from table at a given index

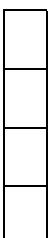
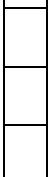
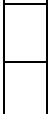
### Instruction Format

op u10

### Syntax Example

EL\_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 EL\_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 EL\_S.

Each instruction executed by the EL\_S instruction from the table pointed to by EI\_BASE is effectively in the execution slot of the EL\_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 EL\_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
- ❑ 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 delay-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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

EI_S	u10	01011 <u>uuuuuuuuuu</u>
------	-----	-------------------------

## ENTER\_S

### Function

Function Prologue 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 prologue 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.

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 - M           // 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

ENTER_S	u6	110000 <u>UU</u> 111 <u>uuuu</u> 0
---------	----	------------------------------------

# 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 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.



**Note** 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
swap( int32_t new_value, int32_t *word ) { /* EX */
    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 8-1 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 8-2 To Release Semaphore using ST**

```
release_resource:  
    MOV R2, 0x00000000  
    ST  R2, [SEMAPHORE_ADDR] ; Release Semaphore using ST  
                                ; indicates semaphore is free  
                                ; release semaphore
```

**Syntax and Encoding**

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

**Instruction Code**

EX<.di> b, [c]	00100bbb00101111DBBBCCCCCC0001100
EX<.di> b, [u6]	00100bbb01101111DBBBuuuuuu0001100

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

EXTB<.f>	b, c	00100 <b>bbb</b> 00101111 <b>FBBB</b> CCCCCC000111
EXTB<.f>	b, u6	00100 <b>bbb</b> 01101111 <b>FBBB</b> uuuuuu000111
EXTB_S	b, c	01111 <b>bbbccc</b> 01111

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

EXTH<.f>	b, c	00100 <b>bbb</b> 00101111 <b>FBBB</b> CCCCCC <b>001000</b>
EXTH<.f>	b, u6	00100 <b>bbb</b> 01101111 <b>FBBB</b> uuuuuu <b>001000</b>
EXTH_S	b, c	01111 <b>bbbccc</b> 10000

## 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	<input type="checkbox"/>	= Unchanged
N	<input type="checkbox"/>	= Unchanged
C	<input type="checkbox"/>	= Unchanged
V	<input type="checkbox"/>	= Unchanged

#### FPU\_STATUS Flags

IV	<input type="checkbox"/>	<ul style="list-style-type: none"> <li>Set when a floating-point number is converted to an integer, where the floating-point number is NaN, infinity, or the conversion result is too big to be represented as an integer.</li> </ul>
		<ul style="list-style-type: none"> <li>or</li> <li>If any of the input operands is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).</li> </ul>
DZ	<input type="checkbox"/>	= Unchanged

OF		= Unchanged
UF		= Unchanged
IX		• Set when result is rounded in case of conversion from an integer to a floating-point number

## 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

Instruction Code		
FCVT32	a,b,c	00110bbb000010000BBBBCCCCCCCCAAAAAA
FCVT32	a,b,u6	00110bbb010010000BBBuuuuuuuuAAAAAA

FCVT32	b, b, s12	00110bbb100010000 BBBssssssSSSSSS
FCVT32<.cc>	b, b, c	00110bbb110010000 BBBCCCCC0QQQQQ
FCVT32<.cc>	b, b, u6	00110bbb110010000 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## FCVT32\_64

### Function

Convert from 32-bit data formats (single-precision float, signed, or unsigned integer) to 64-bit data formats.

### Alias

FINT2D, FUINT2D, FS2L, FS2L\_RZ, FS2UL, FS2UL\_RZ, FS2D

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1)

### Operation

```
A = (long) b;
A = (ulong) b;
A = (double) b;
```



**Note** A is a register pair representing a 64-bit operand, and must be specified as an even numbered register. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A, b, u6

### Syntax Example

FCVT32\_64 A,b,u6

### STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

## FPU\_STATUS Flags

IV	<ul style="list-style-type: none"> <li>• Set when a floating-point number is converted to an integer, where the floating-point number is NaN, infinity, or the conversion result is too big to be represented as a long integer. Or If any of the input operands to a floating-point instruction is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is enabled).</li> </ul>
DZ	= Unchanged
OF	= Unchanged
UF	= Unchanged
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded in case of conversion from an integer to a floating-point number</li> </ul>

## Description

This instruction converts a 32-bit data format into a 64-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
FS2L	FCVT32_64 a, b, 0b000011	Single-precision float to 64-bit integer
FS2L_RZ	FCVT32_64 a, b, 0b001011	Single-precision float to 64-bit integer; round to zero
FS2UL	FCVT32_64 a, b, 0b000001	Single-precision float to 64-bit unsigned integer
FS2UL_RZ	FCVT32_64 a, b, 0b001001	Single-precision float to 64-bit unsigned integer; round to zero
FINT2D	FCVT32_64 a, b, 0b000010	32-bit integer to double-precision float
FUINT2D	FCVT32_64 a, b, 0b000000	32-bit unsigned integer to double-precision float
FS2D	FCVT32_64 a, b, 0b000100	Single-precision float to double-precision float

## Pseudo Code

```
A = (long) (b);                                     /* FCVT32_64 */
```

## Assembly Code Example

```
FCVT32_64 r0,r2,u6      ; conversion from r2 to (r1,r0)
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
FCVT32_64	a,b,c	00110bbb000010010 BBBCCCCC <del>AAAAAA</del>
FCVT32_64	a,b,u6	00110bbb010010010 BBBuuuuuu <del>AAAAAA</del>
FCVT32_64	b,b,s12	00110bbb100010010 BBBssssss <del>SSSSSS</del>
FCVT32_64<.cc>	b,b,c	00110bbb110010010 BBBCCCCC <del>0QQQQQ</del>
FCVT32_64<.cc>	b,b,u6	00110bbb110010010 BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## FCVT64

### Function

Convert between two 64-bit data formats: double-precision float, signed, or unsigned long

### Alias

FD2L, FD2UL, FL2D, FUL2D, FD2L\_RZ, FD2UL\_RZ

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1)

### Operation

```
A = long B;  
A = ulong B;  
A = double B;
```



A and B are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A, B, u6

### Syntax Example

FCVT64 A,B,u6

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

## FPU\_STATUS Flags

IV	<ul style="list-style-type: none"> <li>• Set when a floating-point number is converted to a long integer, where the floating-point number is NaN, infinity, or the conversion result is too big to be represented as a long long.</li> </ul> <p>Or</p> <ul style="list-style-type: none"> <li>If any of the input operands to a floating-point instruction is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is enabled).</li> </ul>
DZ	= Unchanged
OF	= Unchanged
UF	= Unchanged
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded in case of conversion from a long integer to a floating-point number</li> </ul>

## Description

This instruction converts a 64-bit data format into another 64-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
FD2L	FCVT64 a, b, 0b000011	Double-precision float to 64-bit signed integer
FD2L_RZ	FCVT64 a, b, 0b001011	Double-precision float to 64-bit signed integer; round to zero
FL2D	FCVT64 a, b, 0b000010	64-bit integer to double-precision float
FD2UL	FCVT64 a, b, 0b000001	Double-precision float to 64-bit unsigned integer
FD2UL_RZ	FCVT64 a, b, 0b001001	Double-precision float to 64-bit unsigned integer; round to zero
FUL2D	FCVT64 a, b, 0b000000	64-bit unsigned integer to double-precision float

## Pseudo Code

```
A = long B;                                /* FCVT64 */
```

## Assembly Code Example

```
FCVT64 r2,r4,u6      ; conversion from (r5,r4) to (r3,r2)
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

		<b>Instruction Code</b>
FCVT64	a, b, c	00110bb0001110000BBBCCCCC <del>AAAAAA</del> 0
FCVT64	a, b, u6	00110bb0011110000BBBuuuuuu <del>AAAAAA</del> 0
FCVT64	b, b, s12	00110bb0101110000BBBsssss <del>SSSSSS</del>
FCVT64<.cc>	b, b, c	00110bb0111110000BBBCCCCC <del>0QQQQQ</del>
FCVT64<.cc>	b, b, u6	00110bb0111110000BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## FCVT64\_32

### Function

Convert from 64-bit data formats to 32-bit data formats

### Alias

FD2INT, FD2UINT, FL2S, FUL2S, FD2S, FD2INT\_RZ, FD2UINT\_RZ

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1)

### Operation

```
a = int B;
a = uint B;
a = float B;
```



B is a register pair representing 64-bit operand, and should be specified as even numbered register. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#)

### Instruction Format

op a, B, u6

### Syntax Example

FCVT64\_32 a,B,u6

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

## FPU\_STATUS Flags

IV	<ul style="list-style-type: none"> <li>Set when a double is converted to an integer or a double is converted to a single, where the floating-point number is NaN, infinity, or the conversion result is too large to be represented as an integer. Or If any of the input operands to a floating-point instruction is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is enabled).</li> </ul>
DZ	= Unchanged
OF	<ul style="list-style-type: none"> <li>Set when double to single conversion results in a number too large to be represented as a float.</li> </ul>
UF	<ul style="list-style-type: none"> <li>Set when double to single conversion results in a number too small to be represented in single-precision format.</li> </ul>
IX	<ul style="list-style-type: none"> <li>Set when result is rounded</li> </ul>

## Description

This instruction converts a 64-bit data format into a 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
FD2INT	FCVT64_32 a, b, 0b000011	Double-precision float to 32-bit integer
FD2INT_RZ	FCVT64_32 a, b, 0b001011	Double-precision float to 32-bit integer; round to zero
FD2UINT	FCVT64_32 a, b, 0b000001	Double-precision float to 32-bit unsigned integer
FD2UINT_RZ	FCVT64_32 a, b, 0b001001	Double-precision float to 32-bit unsigned integer; round to zero
FL2S	FCVT64_32 a, b, 0b000010	64-bit integer to single-precision float
FUL2S	FCVT64_32 a, b, 0b000000	64-bit unsigned integer to double-precision float
FD2S	FCVT64_32 a, b, 0b000100	Double-precision float to single-precision float

## Pseudo Code

```
a = int B;                                     /* FCVT64_32 */
```

## Assembly Code Example

```
FCVT64_32 r5,r2,u6 ; conversion from (r3,r2)to r5
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
FCVT64_32	a,b,c	00110bb0001110010BBBCCCCC <del>AAAAAA</del> 0
FCVT64_32	a,b,u6	00110bb0011110010BBBuuuuuu <del>AAAAAA</del> 0
FCVT64_32	b,b,s12	00110bb0101110010BBBsssss <del>SSSSSS</del>
FCVT64_32<.cc>	b,b,c	00110bb0111110010BBBCCCCC <del>0QQQQQ</del>
FCVT64_32<.cc>	b,b,u6	00110bb0111110010BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## FDABS

### Function

Double-precision floating-point absolute operation.

### Alias

BCLR r0, r2, 0x1F

### Extension Group

BASELINE

### Operation

B = (double) abs (C) ;



**Note** B and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op B, C

### Syntax Example

FDABS B, C

### STATUS32 Flags Affected

Z	<input type="checkbox"/>	= Unchanged
N	<input type="checkbox"/>	= Unchanged
C	<input type="checkbox"/>	= Unchanged
V	<input type="checkbox"/>	= 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                                /* FDABS */
  dest = abs(src)
```

## Assembly Code Example

```
FDABS r0, r2      ; return the absolute of (r3,r2) into (r1,r0)
```

## Syntax and Encoding

FDABS is supported using an assembler alias. For syntax and encoding information, refer to the BCLR instruction used in the following format:

BCLR r0, r2, 0x1F

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

## FDADD

### Function

Double-precision floating-point addition

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1)

### Operation

```
if (cc) {  
    A = (double) (B + C);  
}
```



#### Note

A, B, and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “Data Formats” on page 80.

### Instruction Format

op A,B,C

### Syntax Example

FDADD A,B,C

FDADD<.cc> A,B,C

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

## FPU\_STATUS Flags

IV	<ul style="list-style-type: none"> <li>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 signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is enabled).</li> </ul>
DZ	Unchanged
OF	<ul style="list-style-type: none"> <li>Set when normalized result exceeds the maximum representable number in a double-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>Set when the result is too small to be represented in a double-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>Set when result is rounded</li> </ul>

## Description

The 64-bit operands B and C are added and the result is stored in the destination register pair.

## Pseudo Code

```
if(cc) {                                     /* FDADD */
    A = (double) (B + C);
}
```

## Assembly Code Example

```
FDADD r0,r2,r4 ; floating-point addition of the register pair(r3,r2) with
;the register pair (r5,r4) and the result is stored in
;(r1,r0).
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

FDADD	a,b,c	00110bb0001100010BBBCCCCC <del>AAAAAA</del> 0
FDADD	a,b,u6	00110bb0011100010BBBuuuuuu <del>AAAAAA</del> 0
FDADD	b,b,s12	00110bb0101100010BBBssssss <del>SSSSSS</del>

FDADD<.cc>	b, b, c	00110bb0111100010 BBBCCCCC0QQQQQ
FDADD<.cc>	b, b, u6	00110bb0111100010 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## FDCMP

### Function

Double-precision floating-point comparison

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1)

### Operation

```
if (cc) {
    compare(B, C);
}
```



#### Note

B and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op B,C

### Syntax Example

FDCMP B,C

FDCMP<.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 the comparison result is unordered and FPU_CTRL.IVE==0 (invalid operation exceptions is enabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

## Description

The 64-bit operand C is subtracted from B, and the core flags are updated.



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.IV = (src1 == sNaN) || (src2 == sNaN);
}
/* FDCMP */

```

## Assembly Code Example

```

FDCMP r0,r2      ; floating-point comparison of (r3,r2) and (r1,r0) 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

FDCMP	b,c	00110bb0001100111BBBCCCCC000000
FDCMP	b,u6	00110bb0011100111BBBuuuuuu000000
FDCMP	b,s12	00110bb0101100111BBBssssssSSSSSS

FDCMP<.cc>	b, c	00110bb0111100111BBBCCCCC0QQQQQ
FDCMP<.cc>	b, u6	00110bb0111100111BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## FDCMPF

### Function

Double-precision floating-point comparison operation with IEEE 754 flag generation. This instruction is similar to the FDCMP instruction in cases when either of the instruction operands is a signaling NaN. The FDCMPF instruction updates the invalid flag (FPU\_STATUS.IV) when either of the operands is a quiet or signaling NaN, whereas, the FDCMP instruction updates the invalid flag (FPU\_STATUS.IV) only when either of the operands is a quiet NaN.

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1)

### Operation

```
if (cc) {
    compare(B, C);
}
```



**Note** B and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op B,C

### Syntax Example

FDCMPF B,C

FDCMPF<.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 enabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

## Description

The 64-bit operand C is subtracted from B, and the core flags are updated.



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);           /* FDCMPF
    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 = V_flag;
}

```

## Assembly Code Example

```

FDCMPF r0,r2      ; floating-point comparison of (r3,r2) and (r1,r0) 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

FDCMPF	b,c	00110bb0001101001BBBCCCCC000000
FDCMPF	b,u6	00110bb0011101001BBBuuuuuu000000

FDCMPF	b, s12	00110bb0101101001BBBssssssSSSSSS
FDCMPF<.cc>	b, c	00110bb0111101001BBBCCCCC0QQQQQ
FDCMPF<.cc>	b, u6	00110bb0111101001BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## FDDIV

### Function

Double-precision floating-point division

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1) && (FPU\_DIV\_OPTION == 1)

### Operation

```
if (cc) {
    A = (double) (B / C);
}
```



#### Note

A, B, and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A,B,C

### Syntax Example

FDDIV A,B,C

FDDIV<.cc> A,B,C

### STATUS32 Flags Affected

#### STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

#### FPU\_STATUS Flags

IV		<ul style="list-style-type: none"> <li>• Set when the operands are in the following combinations: 0/0 or infinity/infinity Or If any of the input operands is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is enabled).</li> </ul>
DZ		<ul style="list-style-type: none"> <li>• Set when the divisor is zero</li> </ul>

OF	•	Set when the normalized result exceeds the maximum representable number in a double-precision format
UF	•	Set when the result is too small to be represented in a double-precision format
IX	•	Set when result is rounded

## Description

The 64-bit operand B is divided by the 64-bit C operand, and the result is stored in the destination register pair.

## Pseudo Code

```
if(cc) {                                     /* FDDIV */
    A = (double) (B / C);
}
```

## Assembly Code Example

```
FDDIV r0,r2,r4      ; floating-point division of (r3,r2) by (r5,r4) and the
; result is stored in (r1,r0).
```

## Syntax and Encoding

For more information about the encodings, see “Key for 32-bit Addressing Modes and Encoding Conventions” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

Instruction Code		
FDDIV	a,b,c	00110bb0001101110BBBCCCCC <del>AAAAAA</del> 0
FDDIV	a,b,u6	00110bb0011101110BBBuuuuuu <del>AAAAAA</del> 0
FDDIV	b,b,s12	00110bb0101100010BBBssssss <del>SSSSSS</del>
FDDIV<.cc>	b,b,c	00110bb0111101110BBBCCCCC <del>0QQQQQ</del>
FDDIV<.cc>	b,b,u6	00110bb0111101110BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## FDMADD

### Function

Double-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\_DP\_OPTION == 1) && (FPU\_FMA\_OPTION == 1)

### Operation

```
if (cc) {  
    A = (double) (ACC + (B * C));  
}
```



A, B, and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A,B,C

### Syntax Example

FDMADD A,B,C

FDMADD<.cc> A,B,C

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

## FPU\_STATUS Flags

IV	<ul style="list-style-type: none"> <li>• Set when the B and C operands are in the following combinations:  <math>0 * \text{infinity}</math>            or  <math>\text{infinity} * 0</math></li> <li>■ Set when the A, B, and C operands are in the following combination:  <math>(B * C) = +\text{infinity}</math> and <math>A = -\text{infinity}</math>            or  <math>(B * C) = -\text{infinity}</math> and <math>A = +\text{infinity}</math></li> <li>■ If any of the input operands is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is enabled).</li> </ul>
DZ	Unchanged
OF	<ul style="list-style-type: none"> <li>• Set when the normalized result exceeds the maximum representable number in a double-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>• Set when the result is too small to be represented in a double-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded after the accumulation</li> </ul>

## Description

The product of the 64-bit operand B and C is computed, added to the accumulator, and the resulting sum is rounded and stored in the destination register pair.



This instruction uses the accumulator as described in “[Extension Core Registers](#)” on page [102](#).

## Pseudo Code

```
if(cc) {                                     /* FDMADD */
    A = (double) (ACC + (B * C));
}
```

## Assembly Code Example

```
FDMADD r0,r2,r4      ; floating-point multiplication of (r5,r4) and (r3,r2)
;the product is accumulated (ACCH,ACCL) and the sum
;is stored in (r1,r0).
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

		<b>Instruction Code</b>
FDMADD	a, b, c	00110bb0001101010BBBCCCCCAAAAAA0
FDMADD	a, b, u6	00110bb0011101010BBBuuuuuuAAAAAA0
FDMADD	b, b, s12	00110bb0101101010BBBssssssSSSSSS
FDMADD<.cc>	b, b, c	00110bb0111101010BBBCCCCC0QQQQQ
FDMADD<.cc>	b, b, u6	00110bb0111101010BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## FDMSUB

### Function

Double-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\_DP\_OPTION == 1) && (FPU\_FMA\_OPTION == 1)

### Operation

```
if (cc) {
    A = (double) (ACC - (B * C));
}
```



A and B are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A,B,C

### Syntax Example

FDMSUB A,B,C

FDMSUB<.cc> A,B,C

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

## FPU\_STATUS Flags

IV	<ul style="list-style-type: none"> <li>• ■ Set when the B and C operands are in the following combinations: 0 * infinity or infinity * 0</li> <li>■ Set when the A, B, and C operands are in the following combination: <math>(B * C) = +\infty</math> and <math>A = +\infty</math> or <math>(B * C) = -\infty</math> and <math>A = -\infty</math></li> <li>■ If any of the input operands is a signalling NaN when <code>FPU_CTRL.IVE==0</code> (invalid operation exceptions is enabled).</li> </ul>
DZ	Unchanged
OF	<ul style="list-style-type: none"> <li>• Set when the normalized result exceeds the maximum representable number in a double-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>• Set when the result is too small to be represented in a double-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded</li> </ul>

## Description

The product of the 64-bit operands B and C is computed, subtracted from the accumulator; the result is rounded and stored in the destination register.



This instruction uses the accumulator as described in “[Extension Core Registers](#)” on page [102](#).

## Pseudo Code

```
if(cc) {                                     /* FDMSUB */
    A = (double) (ACC - (B * C));
}
```

## Assembly Code Example

```
FDMSUB r0,r2,r4      ; floating-point multiplication of (r5,r4) and (r3,r2); the
; product is subtracted from the accumulator (ACCH, ACCL) and
; result is ; stored in (r1,r0).
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

		<b>Instruction Code</b>
FDMSub	a, b, c	00110bb0001101100BBBCCCCCAAAAAA0
FDMSub	a, b, u6	00110bb0011101100BBBuuuuuuAAAAAA0
FDMSub	b, b, s12	00110bb0101101100BBBssssssSSSSSS
FDMSub<.cc>	b, b, c	00110bb0111101100BBBCCCCC0QQQQQ
FDMSub<.cc>	b, b, u6	00110bb0111101100BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## FDMUL

### Function

Double-precision floating-point multiplication

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1)

### Operation

```
if (cc) {  
    A = (double) (B * C);  
}
```



#### Note

A, B, and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A,B,C

### Syntax Example

FDMUL A,B,C

FDMUL<.cc>A,B,C

### STATUS32 Flags Affected

Z	<input type="checkbox"/>	= Unchanged
N	<input type="checkbox"/>	= Unchanged
C	<input type="checkbox"/>	= Unchanged
V	<input type="checkbox"/>	= Unchanged

## FPU\_STATUS Flags

IV	• Set when $(0 * \text{infinity})$ or $(\text{infinity} * 0)$ operations occur Or If any of the input operands is a signalling NaN when $\text{FPU\_CTRL.IVE==0}$ (invalid operation exceptions is enabled).
DZ	= Unchanged
OF	• Set when the normalized result exceeds the maximum representable number in a double-precision format
UF	• Set when the result is too small to be represented in a double-precision format
IX	• Set when result is rounded

## Description

The B and C 64-bit operands are multiplied and the result is stored in the destination register pair.

## Pseudo Code

```
if(cc) {                                     /* FDMUL */
    A = (double) (B * C);
}
```

## Assembly Code Example

```
FDMUL r0,r2,r4 ; floating-point multiplication of (r3, r2) with (r5, r4)
; the result is stored in (r1, r0).
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

FDMUL	a,b,c	00110bb0001100000BBBCCCCC <del>AAAAAA</del> 0
FDMUL	a,b,u6	00110bb0011100000BBBuuuuuu <del>AAAAAA</del> 0
FDMUL	b,b,s12	00110bb0101100000BBBsssss <del>SSSSSS</del>
FDMUL<.cc>	b,b,c	00110bb0111100000BBBCCCCC <del>0QQQQQ</del>
FDMUL<.cc>	b,b,u6	00110bb0111100000BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## FDNEG

### Function

Double-precision floating-point negate operation.

### Alias

BXOR r0, r2, 0x1F

### Extension Group

BASELINE

### Operation

`B = (double) neg(C);`



**Note** B and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

`op B, C`

### Syntax Example

`FDNEG B, C`

### STATUS32 Flags Affected

Z	<input type="checkbox"/>	= Unchanged
N	<input type="checkbox"/>	= Unchanged
C	<input type="checkbox"/>	= Unchanged
V	<input type="checkbox"/>	= 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                                /* FDNEG */  
dest = neg(src)
```

## Assembly Code Example

```
FDNEG r0, r2      ; return the negation of (r3,r2) into (r1,r0).
```

## Syntax and Encoding

FDNEG is supported using an assembler alias. For syntax and encoding information, refer to the BXOR instruction used in the following format:

```
BXOR r0, r2, 0x1F
```

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

## FDSQRT

### Function

Double-precision floating-point square-root

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1) && (FPU\_DIV\_OPTION == 1)

### Operation

```
B = (double) sqrt(C);
```



**Note** B and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op B,C

### Syntax Example

```
FDSQRT B,C
```

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

## FPU\_STATUS Flags

IV	•	Set when the C operand is less than -zero
DZ	=	Unchanged
OF	=	Unchanged
UF	•	Set when the result is too small to be represented in a double-precision format
IX	•	Set when result is rounded

## Description

The 64-bit B operand contains the square-root of the 64-bit C operand.

## Pseudo Code

```
B = (double) sqrt(C); /* FDSQRT */
```

## Assembly Code Example

```
FDSQRT r0,r2 ; floating-point square-root of (r3,r2) is stored in (r1,r0)
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

FDSQRT	b,c	00110bb0001011110BBBCCCCC000001
FDSQRT	b,u6	00110bb0011011110BBBuuuuuu000001

## FDSUB

### Function

Double-precision floating-point subtraction

### Extension Group

(HAS\_FPU == 1) && (FPU\_DP\_OPTION == 1)

### Operation

```
if (cc) {
    A = (double) (B - C);
}
```



#### Note

B and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A,B,C

### Syntax Example

FDSUB A,B,C

FDSUB<.cc> A,B,C

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

## FPU\_STATUS Flags

IV	<ul style="list-style-type: none"> <li>• Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is enabled).</li> </ul>
DZ	Unchanged
OF	<ul style="list-style-type: none"> <li>• Set when the normalized result exceeds the maximum representable number in a double-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>• Set when the result is too small to be represented in a double-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded</li> </ul>

## Description

The 64-bit operand C is subtracted from the 64-bit operand B, and the result is stored in the destination register pair.

## Pseudo Code

```
if(cc) {                                     /* FDSUB */
    A = (double) (B - C);
}
```

## Assembly Code Example

```
FDSUB r0,r2,r4      ; floating-point subtraction of (r5,r4) from (r3,r2) and
; the result is stored in (r1,r0).
```

## Syntax and Encoding

For more information about the encodings, see “Key for 32-bit Addressing Modes and Encoding Conventions” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

FDSUB	a,b,c	00110bb0001100100BBBCCCCC <del>AAAAAA</del> 0
FDSUB	a,b,u6	00110bb0011100100BBBuuuuuu <del>AAAAAA</del> 0
FDSUB	b,b,s12	00110bb0101100100BBBsssss <del>SSSSSS</del>
FDSUB<.cc>	b,b,c	00110bb0111100100BBBCCCCC <del>0QQQQQ</del>
FDSUB<.cc>	b,b,u6	00110bb0111100100BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

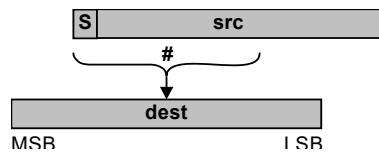
Find First Set

### Extension Group

BITSCAN\_OPTION

### Operation

$\text{dest} \leftarrow \text{bit position of first 1 in source operand, from bit 0 upwards}$



### 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

FFS<.f> b,c	00101bbb00101111FBBBBCCCCC010010
FFS<.f> b,u6	00101bbb01101111FBBBuuuuuu010010

## 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.

The related KFLAG instruction can be used in the normal kernel mode to modify the AE, IE, and RB bits, in addition to the ones modified by the FLAG instruction.

## Pseudo Code

```
if ((src[0] == 1) && (STATUS32[7] == 0)) then          /* FLAG
    STATUS32[0] = 1
    Halt()
else
    STATUS32[11:8] = src[11:8]
    if (STATUS32[7] == 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]
```

```

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*/

```

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
FLAG	c	00100 <b>rrr</b> 001010010RRRCCCCC <b>RRRRRR</b>
FLAG	u6	00100 <b>rrr</b> 011010010RRRuuuuuu <b>RRRRRR</b>
FLAG	s12	00100 <b>rrr</b> 101010010RRRsaaaaa <b>SSSSSS</b>
FLAG<.cc>	c	00100 <b>rrr</b> 111010010RRRCCCCC <b>0QQQQQ</b>
FLAG<.cc>	u6	00100 <b>rrr</b> 111010010RRRuuuuuu <b>1QQQQQ</b>

## FLS

### Function

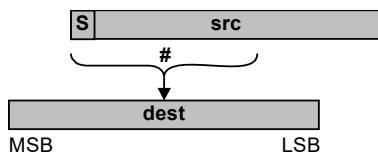
Find Last Set

### Extension Group

BITSCAN\_OPTION

### Operation

$\text{dest} \leftarrow \text{bit position of last 1 in source operand, from bit 0 upwards}$



### 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. The returned value for a source operand of zero is 0. Examples of returned values are shown in the following table. The '-' symbol signifies any hexadecimal digit:

Operand Value	Result	Z	N
0x8-----	31	0	1
0x4-----	30	0	0
0x2-----	29	0	0

Operand Value	Result	Z	N
:	:	:	:
0x00000004	2	0	0
0x00000002	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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

FLS<.f> b,c	00101bbb00101111FBBBCCCCCC010011
FLS<.f> b,u6	00101bbb01101111FBBBuuuuuu010011

## FSABS

### Function

Single-precision floating-point absolute operation.

### Alias

BCLR r1, r3, 0x1F

### Extension Group

BASELINE

### Operation

```
b = (float) 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 assembler 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

## 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	<ul style="list-style-type: none"> <li>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 signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).</li> </ul>
DZ	Unchanged
OF	<ul style="list-style-type: none"> <li>Set when the normalized result exceeds the maximum representable number in a single-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>Set when the result is too small to be represented in a single-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>Set when result is rounded</li> </ul>

## 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

Instruction Code			
FSADD	a,b,c	00110	bbb00000010 BBBCCCCCAAAAAA
FSADD	a,b,u6	00110	bbb010000010 BBBuuuuuuAAAAAA
FSADD	b,b,s12	00110	bbb100000010 BBBssssssSSSSSS
FSADD<.cc>	b,b,c	00110	bbb110000010 BBBCCCCC0QQQQQ

FSADD<.cc>	b , b , u6	00110 <b>bbb</b> 110000010 <b>BBB</b> uuuuuu <u>1</u> <b>QQQQQ</b>
------------	------------	--

## 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:

```
ADD.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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

FSCMP	b,c	00110bbb000000111BBBCCCCC000000
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:

```
ADD.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 quiet 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.



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);           /* FSCMPF */
    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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

FSCMPF	b,c	00110bbb00001001BBBCCCCC000000
FSCMPF	b,u6	00110bbb010001001BBBuuuuuu000000
FSCMPF	b,s12	00110bbb100001001BBBsssssSSSSSS

FSCMPF<.cc>	b, c	00110bbb110001001 BBBCCCCC0QQQQQ
FSCMPF<.cc>	b, u6	00110bbb110001001 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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	<ul style="list-style-type: none"> <li>Set when the operands are in the following combinations: 0/0 or infinity/infinity Or If any of the input operands is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).</li> </ul>
DZ	<ul style="list-style-type: none"> <li>Set when the divisor is zero</li> </ul>
OF	<ul style="list-style-type: none"> <li>Set when the normalized result exceeds the maximumrepresentable number in a single-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>Set when the result is too small to be represented in a single-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>Set when result is rounded</li> </ul>

## 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

Instruction Code		
FSDIV	a,b,c	00110bbb000001110 BBBCCCCCAAAAAA
FSDIV	a,b,u6	00110bbb010001110 BBBuuuuuuAAAAAA
FSDIV	b,b,s12	00110bbb100000010 BBBssssssSSSSSS
FSDIV<.cc>	b,b,c	00110bbb110001110 BBBCCCCC0QQQQQ

FSDIV&lt;.cc&gt;

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:

```
ADD.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	<ul style="list-style-type: none"> <li>• Set when the b and c operands are in the following combinations:            0 * infinity            or            infinity * 0         </li> <li>■ Set when the a, b, and c operands are in the following combinations:  <math>(b * c) = +\infty</math> and <math>a = -\infty</math>            or  <math>(b * c) = -\infty</math> and <math>a = +\infty</math> </li> <li>■ If any of the input operands is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).</li> </ul>
DZ	Unchanged
OF	<ul style="list-style-type: none"> <li>• Set when the normalized result exceeds the maximum representable number in a single-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>• Set when the result is too small to be represented in a single-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded after the accumulation</li> </ul>

## 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.



This instruction uses the accumulator as described in “[Extension Core Registers](#)” on page [102](#).

## Pseudo Code

```
if(cc) {                                     /* FSMADD */
    a = (float) (acc + (b * c));
}
```

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
FSMADD	a, b, c	00110bbb00001010 BBBCCCCCAAAAAA
FSMADD	a, b, u6	00110bbb010001010 BBBuuuuuuAAAAAA
FSMADD	b, b, s12	00110bbb100001010 BBBssssssSSSSSS
FSMADD<.cc>	b, b, c	00110bbb110001010 BBBCCCCC0QQQQQ
FSMADD<.cc>	b, b, u6	00110bbb110001010 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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	<input type="checkbox"/>	= Unchanged
N	<input type="checkbox"/>	= Unchanged
C	<input type="checkbox"/>	= Unchanged
V	<input type="checkbox"/>	= Unchanged

## FPU\_STATUS Flags

IV	<ul style="list-style-type: none"> <li>• Set when the b and c operands are in the following combinations:            0 * infinity            or            infinity * 0         </li> <li>■ Set when the a, b, and c operands are in the following combinations:  <math>(b * c) = +\text{infinity}</math> and <math>a = +\text{infinity}</math>            or  <math>(b * c) = -\text{infinity}</math> and <math>a = -\text{infinity}</math> </li> <li>■ If any of the input operands is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).</li> </ul>
DZ	Unchanged
OF	<ul style="list-style-type: none"> <li>• Set when the normalized result exceeds the maximum representable number in a single-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>• Set when the result is too small to be represented in a single-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded</li> </ul>

## Description

The product of b and c operands is computed, subtracted from the accumulator; the result is rounded and stored in the destination register.



This instruction uses the accumulator as described in “[Extension Core Registers](#)” on page [102](#).

## Pseudo Code

```
if(cc) {                                     /* FSMSUB */
    a = (float) (acc - (b * c));
}
```

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
FSMSUB	a, b, c	00110 <b>bbb</b> 000001100 <b>BBB</b> CCCCC <del>AAAAAA</del>
FSMSUB	a, b, u6	00110 <b>bbb</b> 010001100 <b>BBB</b> uuuuuu <del>AAAAAA</del>
FSMSUB	b, b, s12	00110 <b>bbb</b> 100001100 <b>BBB</b> ssssss <del>SSSSSS</del>
FSMSUB<.cc>	b, b, c	00110 <b>bbb</b> 1100001100 <b>BBB</b> CCCCC <del>0QQQQQ</del>
FSMSUB<.cc>	b, b, u6	00110 <b>bbb</b> 1100001100 <b>BBB</b> uuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

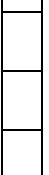
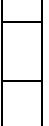
op a,b,c

### 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	<ul style="list-style-type: none"> <li>• Set when <math>(0 * \text{infinity})</math> or <math>(\text{infinity} * 0)</math> operations occur Or If any of the input operands is a signalling NaN when <math>\text{FPU\_CTRL.IVE==0}</math> (invalid operation exceptions is not enabled).</li> </ul>
DZ	= Unchanged
OF	<ul style="list-style-type: none"> <li>• Set when the normalized result exceeds the maximum representable number in a single-precision format</li> </ul>
UF	<ul style="list-style-type: none"> <li>• Set when the result is too small to be represented in a single-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded</li> </ul>

## Description

The b and c operands are multiplied and the product is stored in the destination register.

## Pseudo Code

```
if(cc) {                                     /* FSMUL */
    a = (float) (b * c);
}
```

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

<b>Instruction Code</b>		
FSMUL<.f>	a, b, c	00110bbb000000000BBBCCCCC <del>AAAAAA</del>
FSMUL<.f>	a, b, u6	00110bbb010000000BBBuuuuuu <del>AAAAAA</del>
FSMUL<.f>	b, b, s12	00110bbb100000000BBBssssss <del>SSSSSS</del>
FSMUL<.cc>	b, b, c	00110bbb110000000BBBCCCCC0QQQQQ
FSMUL<.cc>	b, b, u6	00110bbb110000000BBBuuuuuu1 <del>QQQQQ</del>

## 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:

```
ADD.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 = (float) 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 assembler 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

## 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	<ul style="list-style-type: none"> <li>• Set when the c operand is less than -zero Or If any of the input operands is a signalling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).</li> </ul>
DZ	= Unchanged
OF	= Unchanged
UF	<ul style="list-style-type: none"> <li>• Set when the result is too small to be represented in a single-precision format</li> </ul>
IX	<ul style="list-style-type: none"> <li>• Set when result is rounded</li> </ul>

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### 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 signalling 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

FSSUB	a,b,c	00110bbb00000100 BBBCCCCC <del>AAAAAA</del>
FSSUB	a,b,u6	00110bbb010000100 BBBuuuuuu <del>AAAAAA</del>
FSSUB	b,b,s12	00110bbb100000100 BBBssssss <del>SSSSSS</del>
FSSUB<.cc>	b,b,c	00110bbb110000100 BBBCCCCC <del>0QQQQQ</del>
FSSUB<.cc>	b,b,u6	00110bbb110000100 BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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 5-37](#) on page [285](#) provides more information about the delay slot modes, .d.

[Table 5-7](#) on page [260](#) 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

J	[c]	00100RRR00100000RRRRCCCCCRRRRRR
J	u6	00100RRR01100000RRRUuuuuuuRRRRRR

J	s12	00100RRR1010000RRRssssssSSSSSS
J.D	[c]	00100RRR00100001RRRCCCCCRRRRRR
J.D	u6	00100RRR01100001RRRuuuuuuRRRRRR
J.D	s12	00100RRR10100001RRRssssssSSSSSS
J_S	[b]	0111bbb00000000
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 5-37](#) on page [285](#) provides more information about the delay slot modes, .d.

[Table 5-7](#) on page [260](#) 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

Jcc	[c]	00100RRR11100000RRRRCCCCC0QQQQQ
Jcc	u6	00100RRR11100000RRRuuuuuu1QQQQQ

Jcc.D	[c]	00100RRR11100001RRRRCCCCC0QQQQQ
Jcc.D	u6	00100RRR11100001RRRuuuuuu1QQQQQ
JEQ_S	[BLINK]	0111110011100000
JNE_S	[BLINK]	0111110111100000

## JL

### Function

Jump and Link unconditionally

### Extension Group

BASELINE

### Operation

$\text{PC} = c; \text{BLINK} = \text{NEXT\_PC};$

### 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 5-37](#) on page [285](#) provides more information about the delay slot modes, .d.

[Table 5-7](#) on page [260](#) 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

JL	[c]	00100RRR00100010RRRRCCCCCRRRRRR
JL	u6	00100RRR01100010RRRUuuuuuRRRRRR

JL	s12	00100RRR10100010RRRRssssssSSSSSS
JL.D	[c]	00100RRR00100011RRRRCCCCCRRRRRR
JL.D	u6	00100RRR01100011RRRRuuuuuuRRRRRR
JL.D	s12	00100RRR10100011RRRRssssssSSSSSS
JL_S	[b]	01111bbb0100000
JL_S.D	[b]	01111bbb01100000

## JLcc

### Function

Jump and Link Conditionally

### Extension Group

BASELINE

### Operation

```
if (cc) {PC = c; BLINK = NEXT_PC;}
```

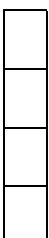
### Instruction Format

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 8-2 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 5-37](#) on page [285](#) provides more information about the delay slot modes, .d.

[Table 5-7](#) on page [260](#) 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

**Instruction Code**

JLcc	[c]	00100RRR11100010RRRCCCCC0QQQQQ
JLcc	u6	00100RRR11100010RRRuuuuuu1QQQQQ
JLcc.D	[c]	00100RRR11100011RRRCCCCC0QQQQQ
JLcc.D	u6	00100RRR11100011RRRuuuuuu1QQQQQ

## JLI\_S

### Function

Jump and Link Indexed

### Extension Group

CODE\_DENSITY option 3

### Operation

$\text{BLINK} = \text{next\_PC}; \text{PC} = \text{JLI\_BASE} + (\text{u10} \ll 2);$

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

```
JLI_S u10 010110uuuuuuuuuuuu
```

# 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)           /* 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]
```

```

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 */

```

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
KFLAG	c	00100RRR001010011RRRCCCCCRRRRR
KFLAG	u6	00100RRR011010011RRRuuuuuuRRRRR
KFLAG	s12	00100RRR101010011RRRssssssSSSSS
KFLAG<.cc>	c	00100RRR111010011RRRCCCCC0QQQQQ
KFLAG<.cc>	u6	00100RRR111010011RRRuuuuuu1QQQQQ

## LD LDH LDW LDB LDD LDL

### Function

Load from Memory



The LDW mnemonic is deprecated, and equivalent to LDH

### Extension Group

LD, LDH, LDB: BASELINE

LDD: LL64\_OPTION

LDL: ARC64\_ISA

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 5-13](#) on page 263 lists the data size modes.

Data can be sign-extended by enabling sign extend <.x>. [Table 5-14](#) on page 264 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 5-12](#) on page 263 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 5-10](#) on page 262 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.

A Load Double (LDD) instruction reads an 8-byte double-word from memory and writes the data to a pair of core registers. The effective address given in the instruction is used to load a data word into the even register Rn, and the address +4 is used to load a data word into the odd register Rn+1.

For a discussion about how the LDD works with 64-bit operands, see “[64-Bit Data](#)” on page 88.

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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

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]	00100bbbba110ZZXDBBBCCCCCBBBBBAAAAAA
LD_S	a, [b,c]	01100bbbccc00aaa
LD_S	b, [SP,u7]	11000bbb000uuuuuu
LD_S	b, [PCL,u10]	11010bbbuuuuuuuuuu
LD_S	c, [b,u7]	10000bbbcccuuuuuu
LD_S	R0, [GP,s11]	1100100ssssssssss
LDB_S	a, [b,c]	01100bbbccc01aaa
LDB_S	b, [SP,u7]	11000bbb001uuuuuu

LDB_S	c, [b, u5]	10001bbbccc <u>uuuuu</u>
LDB_S	R0, [GP, s9]	1100101sssssssss
LDH_S	a, [b, c]	01100bbbccc <u>10aaa</u>
LDH_S	c, [b, u6]	10010bbbccc <u>uuuuu</u>
LDH_S	R0, [GP, s10]	1100110sssssssss
LDH_S.X	c, [b, u6]	10011bbbccc <u>uuuuu</u>



**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

LD<zz><.x><.aa><.di>	0, [b, s9]	00010bbbssssssssSBBBDaaZZX111110
LD<zz><.x><.aa><.di>	0, [b, c]	00100bbbbaa110ZZXDBBBCCCCC111110

### Encodings supported by the CODE DENSITY options

LD_S	R0, [h, u5]	01000U00hhhuu1HH
LD_S	R1, [h, u5]	01000U01hhhuu1HH
LD_S	R2, [h, u5]	01000U10hhhuu1HH
LD_S	R3, [h, u5]	01000U11hhhuu1HH
LD_S.AS	a, [b, c]	01001bbbccc <u>00aaa</u>
LD_S	R1, [GP, s11]	01010SSSSSS00sss

### Encodings supported by LL64\_OPTION

Instruction Code		
LDD<.aa><.di>	a, [b]	00010bbb000000000BBDaa110AAAAAA0
LDD<.aa><.di>	a, [b, s9]	00010bbbssssssssSBBBDaa110AAAAAA0
LDD<.aa><.di>	a, [b, c]	00100bbbbaa110110DBBBCCCCCAAAAA0

## 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:

```
ADD.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

```
addr = AUX_LDI_BASE + (src2 << 2); b = *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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
LDI	b, [c]	00100bbb00100110RBBBCCCCCRRRRRR
LDI	b, [u6]	00100bbb01100110RBBBuuuuuuRRRRRR
LDI	b, [s12]	00100bbb10100110RBBBssssssSSSSSS
LDI_S	b, u7	01010bbbUUUU1uuu

## LEAVE\_S

### Function

Function epilogue 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 epilogue 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.

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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

LEAVE_S	u7	11000 <u>UUU</u> 110 <u>uuuu</u> 0
---------	----	------------------------------------

## LLOCK

### Function

Load Locked

### Extension Group

ATOMIC\_OPTION == 1

### Operation

`_llock(b,c);`

### Instruction Format

`op b,c`

### Syntax Example

`LLOCK<.di> b, [c]`

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

The 32-bit word, located at the address given by the source operand, is loaded from the data address space into the destination register. If the load operation completes, the equivalent physical address is saved in the non-architecturally visible Lock Physical Address (LPA) register, and the non-architecturally visible Lock Flag (LF) register is set to 1.

The LF register is cleared whenever an exception or interrupt is taken, or whenever the processor completes a store instruction to the address contained in the LPA register.

If any external agent, such as another processor, completes a store to the address contained in the LPA register, the LF register is cleared. Such writes, and the clearing of LF, are an atomic action.

In a virtual memory environment, the LLOCK instruction does not check the write permissions on the operand location. Therefore, the following SCOND instruction may fail because of insufficient privileges even if the LLOCK instruction completes its read successfully.

The LLOCK.DI instruction operates directly on external memory, bypassing any data cache that may be present in the path from the processor to the physical memory system. This form must be used in

implementations of the ARCv2 that do not provide hardware cache coherency for shared memory accesses.

The memory read operations implied by the LLOCK instruction, without the .DI modifier, are cached reads. This form of the instruction must be used only in implementations of the ARCv2 that provide hardware cache coherency for shared memory accesses, or in systems with a single processor.

The effective address of LLOCK must be aligned to a 4-byte boundary, otherwise an EV\_Misaligned exception is raised even if STATUS32.AD==1.

## Pseudo Code

```
EA  = PhysicalAddress(c); translate address from c if
      ; VM enabled
b   = ReadWord(EA[31:2]); read 32-bit word from memory into b
LPA = EA[31:2]           ; for non-PAE builds; PAE builds use EA[39:2]
LF  = 1
```

## Assembly Code Example

```
LLOCK r1, [r2]    ; Load from address given by r2 and set LF to 1 Set
                  ; LPA to r2
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

LLOCK<.di>	b, [c]	00100bbb00101111DBBBCCCCC010000
LLOCK<.di>	b, [u6]	00100bbb01101111DBBBuuuuuu010000

## LLOCKD

### Function

Load locked on 64-bit data

### Extension Group

(ATOMIC\_OPTION == 1) and (LL64\_OPTION==1)

### Operation

\_llockd(b,c);

### Instruction Format

op B,C

### Syntax Example

LLOCKD<.di> B, [C]

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

The LLOCKD instruction is similar to the [LLOCK](#) instruction except that LLOCKD operates on 64-bit data objects and loads 64-bit values into a pair of 32-bit core registers.

The effective address of LLOCKD must be aligned to an eight-byte boundary. Otherwise, an EV\_Misaligned exception is raised even if STATUS32.AD==1.

### Pseudo Code

```

EA  = PhysicalAddress(C)      ; translate address from C if
                                ; VM enabled
B  = ReadDoubleWord(EA[31:3]); read double-word(64-bit) from
                                ; memory into B
LPA = EA[31:3]                ; for non-PAE builds; PAE builds use EA[39:3]
LF  = 1

```

## Assembly Code Example

```
LLOCKD r0, [r2] ; Load from address given by r2 to the register pair  
; (r0,r1) and set LF to 1. Set LPA to r2
```

### Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

#### Instruction Code

LLOCKD<.di>	b, [c]	00100bbb00101111DBBBCCCCCC010010
LLOCKD<.di>	b, [u6]	00100bbb01101111DBBBuuuuuu010010

## LPcc

### 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 maybe 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	aux_reg[LP_END] = PCL + u7 aux_reg[LP_START] = NEXT_PC	PC = PCL + u7
LP s13	aux_reg[LP_END] = PCL + s13 aux_reg[LP_START] = NEXT_PC	Not possible

[Table 5-7](#) on page [260](#) 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 (0xFFFFFFF or 0xFFFFFFFF) 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.

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 8-3 Example Loop Code

```

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 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:

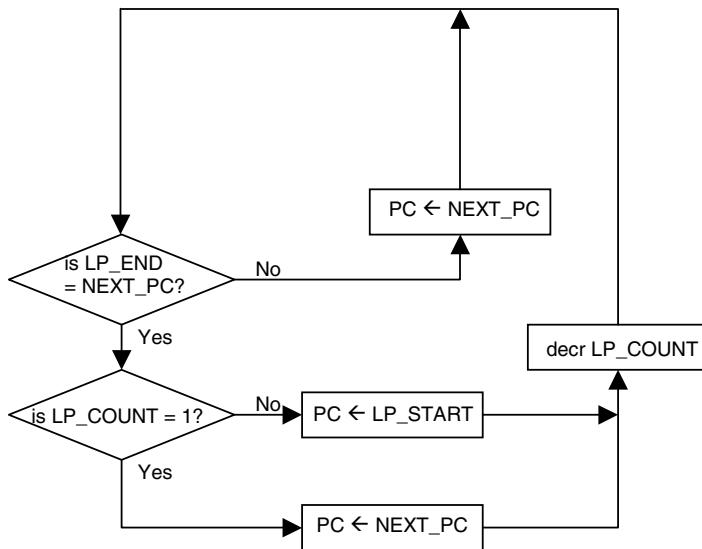
- ❑ STATUS32[DE] is 0, indicating that the instruction is not in the delay slot of a branch, or the branch was not taken.
- ❑ STATUS32[L] is 0
  - This bit is set to 1 to disable loop-end detection.
- ❑ The instruction to be completed is the last in a loop
  - PC + current instruction size = LP\_END
- ❑ 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) >= 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 8-1](#).

**Figure 8-1 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 8-3](#). This table covers loop setup and use of long immediate data.



**Note** 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 8-3 When Reads and Writes Take Effect**

	<b>Instruction</b>	<b>Writing Takes Effect</b>	<b>Reading Uses</b>	<b>Writing Takes Effect</b>	<b>Reading Uses</b>
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	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 8-4](#).

### Example 8-4 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

#### Instruction Code

LP<cc>	u7	00100RRR11101000RRRRuuuuuu1QQQQQ
LP	u7	00100RRR01101000RRRRuuuuuuRRRRRR
LP	s13	00100RRR10101000RRRssssssSSSSSS

## LR

### Function

Load from Auxiliary Register

### Extension Group

BASELINE

### Operation

$b = \text{lr}(c);$

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

LR	b, [c]	00100bbb00101010RBBBCCCCCCRRRRRR
LR	b, [u6]	00100bbb01101010RBBBuuuuuuRRRRRR
LR	b, [s12]	00100bbb10101010RBBBssssssSSSSSS

## LSL16

### Function

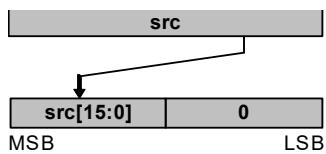
Logical Shift Left 16

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

LSL16<.f>	b,c	00101bbb00101111FBBBCCCCC001010
LSL16<.f>	b,u6	00101bbb01101111FBBBuuuuuu001010

## LSL8

### Function

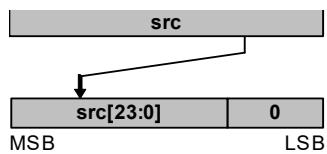
Logical Shift Left 8

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

LSL8<.f> b,c 00101bbb00101111FBBBBCCCCC001111

LSL8<.f> b,u6 00101bbb01101111FBBBuuuuuu001111

## LSR

### Function

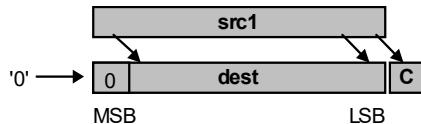
Logical Shift Right

### Extension Group

BASELINE

### Operation

$b = (\text{unsigned}) c >> 1;$



### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

LSR<.f>	b, c	00100bbb00101111FBBBCCCCC000010
LSR<.f>	b, u6	00100bbb01101111FBBBuuuuuuu000010
LSR_S	b, c	01111bbbccc11101

## LSR multiple

### Function

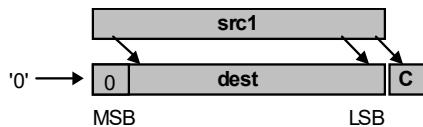
Multiple Logical Shift Right

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
LSR<.f>	a,b,c	00101bbb00000001FBBBBCCCCCAAAAAAA
LSR<.f>	a,b,u6	00101bbb01000001FBBBuuuuuuAAAAAAA
LSR<.f>	b,b,s12	00101bbb10000001FBBBssssssSSSSSS
LSR<.cc><.f>	b,b,c	00101bbb11000001FBBBCCCCCC0QQQQQ
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:

```
ADD.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

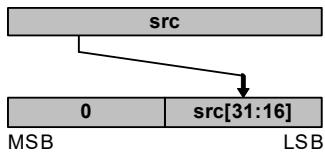
Logical Shift Right 16

### Extension Group

SWAP\_OPTION

### Operation

$b = c >> 16;$



### Instruction Format

op b,c

### Syntax Example

LSR16<.f> b,c

### STATUS32 Flags Affected

Z	<input checked="" type="checkbox"/>	= Set if result is zero
N	<input checked="" type="checkbox"/>	= Set if most-significant bit of result is set
C	<input type="checkbox"/>	= Unchanged
V	<input type="checkbox"/>	= 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

LSR16<.f> b,c 00101bbb00101111FBBCCCCCC001011

LSR16<.f> b,u6 00101bbb01101111FBBCuuuuuu001011

## LSR8

### Function

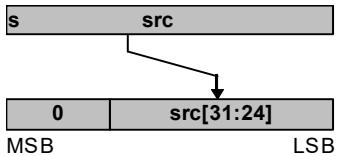
Logical Shift Right 8

### 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	•	= Set if most-significant bit of result is set
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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

LSR8<.f> b,c 00101bbb00101111FBBBBCCCCC001110

LSR8<.f> b,u6 00101bbb01101111FBBBuuuuuu001110

## MAC

This section describes the standard MAC instruction.

### Function

Signed 32x32 multiplication and accumulation.

### Extension Group

MPY\_OPTION > 6

### Operation

```
if (cc) {
    result = acc + (b * c);
    a = result.w0;
    acc = result;
}
```



**Note** wo represents the lower 32-bits of a 64-bit result. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### 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 accumulator to form the result. The least-significant 32 bits of the 64-bit result are assigned to the destination register, and the entire 64-bit result is assigned to the accumulator.

Flags are updated only if the set-flags suffix (.F) is used. The overflow (V) flag is set if an overflow occurs during accumulation. The MAC instruction never clears the V flag. The sign flag (N) is set to the sign of the accumulator result.

## Pseudo Code

```

if(cc) {
    result = acc +(b * c);
    a = result.w0;
    acc = result;
}
/* MAC*/

```

## Assembly Code Example

```

MAC r1,r2,r3      ; 32x32 multiply and accumulate r2 and r3 and store result
                   ; in r1

```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#). For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

MAC<.f>	a,b,c	00101bbb00001110FBBBCCCCCCCCAAAAAA
MAC<.f>	a,b,u6	00101bbb01001110FBBBuuuuuuAAAAAA
MAC<.f>	b,b,s12	00101bbb10001110FBBBssssssSSSSSS
MAC<.cc><.f>	b,b,c	00101bbb11001110FBBBCCCCCC0QQQQQ
MAC<.cc><.f>	b,b,u6	00101bbb11001110FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

This section describes the standard MACD instruction.

### Function

Signed 32x32 multiplication and accumulation. Returns a 64-bit result.

### Extension Group

MPY\_OPTION > 7

### Operation

```
if (cc) {
    result = acc + (b * c);
    A = result;
    acc = result;
}
```



**Note** A is a register pair representing a 64-bit operand, and should be specified as an even numbered register. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A, b, c

### Syntax Example

MACD <.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

Compute the 32x32 product of the first and second source operands. The product is added to the accumulator to form the result. The result is then assigned to the destination register pair and the accumulator.

Flags are updated only if the set-flags suffix (.F) is used. The overflow (V) flag is set if an overflow occurs during accumulation. The MACD instruction never clears the V flag. The sign flag (N) is set to the sign of the accumulator result.

## Pseudo Code

```

if(cc) {
    result = acc + (b * c);
    A = result;
    acc = result;
}
/* MACD */

```

## Assembly Code Example

```

MACD r0,r2,r3      ; 32x32y64 multiply-accumulate r2 and r3 and store the
                     ; result in (r1,r0)

```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#). For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

MACD<.f>	a,b,c	00101bbb00011010FBBBCCCCCCCCAAAAAA
MACD<.f>	a,b,u6	00101bbb01011010FBBBuuuuuuAAAAAA
MACD<.f>	b,b,s12	00101bbb10011010FBBBssssssSSSSSS
MACD<.cc><.f>	b,b,c	00101bbb11011010FBBBCCCCCCC0QQQQQ
MACD<.cc><.f>	b,b,u6	00101bbb11011010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

This section describes the standard MACDU instruction.

### Function

Unsigned 32x32 multiplication and accumulation. Return a 64-bit result.

### Extension Group

MPY\_OPTION > 7

### Operation

```
if (cc) {
    result = acc + (b * c);
    A = result;
    acc = result;
}
```



**Note** A is a register pair representing a 64-bit operand, and should be specified as an even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A, b, c

### Syntax Example

MACDU <.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

Compute the unsigned 32x32 product of the first and second source operands. The product is added to the accumulator to form the result. The unsigned 64-bit result is then assigned to the destination register pair and the accumulator.

Any flag updates occur only if the set-flags suffix (.F) is used. The overflow (V) flag is set if an overflow occurs during accumulation. This instruction never clears the V flag.

## Pseudo Code

```
if(cc) {
    result = acc + (b * c);
    A = result;
    acc = result;
}
```

## 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



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).  
 For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

MACDU<.f>	a, b, c	00101bbb00011011FBBBCCCCCCCCAAAAAA
MACDU<.f>	a, b, u6	00101bbb01011011FBBBuuuuuuAAAAAA
MACDU<.f>	b, b, s12	00101bbb10011011FBBBssssssSSSSSS
MACDU<.cc><.f>	b, b, c	00101bbb11011011FBBBCCCCCCCC0QQQQQ
MACDU<.cc><.f>	b, b, u6	00101bbb11011011FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Operation

```
if (cc) {
    result = acc + (b * c);
    a = result.w0;
    acc = result;
}
```



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

Compute the unsigned 32x32 product of the first and second source operands. This 64-bit product is added to the accumulator to form the result. The least-significant 32 bits of the 64-bit result are assigned to the destination register, and the 64-bit result is assigned to the accumulator.

Flags are updated only if the set-flags suffix (.F) is used. The overflow (V) flag is set if an overflow occurs during accumulation. The MACU instruction never clears the V flag.

## Pseudo Code

```

if(cc) {
    result = acc + (b * c); /* MACU */
    a = result.w0;
    acc = result;
}

```

## Assembly Code Example

```

MACU r1,r2,r3 ; 32x32 unsigned multiplication
; and accumulation of r2 and r3
; and store result in r1

```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
MACU<.f>	a,b,c	00101bbb00001111FBBBCCCCCCCCAAAAAA
MACU<.f>	a,b,u6	00101bbb01001111FBBBuuuuuuuuAAAAAA
MACU<.f>	b,b,s12	00101bbb10001111FBBBssssssSSSSSS
MACU<.cc><.f>	b,b,c	00101bbb11001111FBBBCCCCCC0QQQQQ
MACU<.cc><.f>	b,b,u6	00101bbb11001111FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## 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 ( $\text{src2} \geq \text{src1}$ )
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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

Instruction Code		
MAX<.f>	a,b,c	00100bbb00001000FBBCCCCCCAAAAAA
MAX<.f>	a,b,u6	00100bbb01001000FBBCuuuuuuAAAAAA
MAX<.f>	b,b,s12	00100bbb10001000FBBCssssssSSSSSS
MAX<.cc><.f>	b,b,c	00100bbb11001000FBBCCCCCC0QQQQQ
MAX<.cc><.f>	b,b,u6	00100bbb11001000FBBCuuuuuu1QQQQQ

## 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:

```
ADD.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 ( $\text{src2} \leq \text{src1}$ )
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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

Instruction Code		
MIN<.f>	a,b,c	00100 <b>bbb</b> 00001001 <b>FBBB</b> CCCCCCAAAAAA
MIN<.f>	a,b,u6	00100 <b>bbb</b> 01001001 <b>FBBB</b> uuuuuuAAAAAA
MIN<.f>	b,b,s12	00100 <b>bbb</b> 10001001 <b>FBBB</b> ssssssSSSSSS
MIN<.cc><.f>	b,b,c	00100 <b>bbb</b> 11001001 <b>FBBB</b> CCCCCC0QQQQQ
MIN<.cc><.f>	b,b,u6	00100 <b>bbb</b> 11001001 <b>FBBB</b> uuuuuu1QQQQQ

## 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:

```
ADD.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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
MOV<.f>	b, c	00100 <b>bbb</b> 00001010F <b>BBB</b> CCCCC <b>RRRRR</b> R
MOV<.f>	b, u6	00100 <b>bbb</b> 01001010F <b>BBB</b> uuuuuu <b>RRRRR</b> R
MOV<.f>	b, s12	00100 <b>bbb</b> 10001010F <b>BBB</b> ssssss <b>SSSSS</b> S
MOV<.cc><.f>	b, c	00100 <b>bbb</b> 11001010F <b>BBB</b> CCCCC <b>0QQQQ</b> Q
MOV<.cc><.f>	b, u6	00100 <b>bbb</b> 11001010F <b>BBB</b> uuuuuu <b>1QQQQ</b> Q
MOV_S	h, s3	01110 <b>sss</b> hhh <b>011HH</b>
MOV_S	0, s3	01110 <b>sss</b> 11001111
MOV_S.NE	b, h	01110 <b>bbb</b> hhh <b>111HH</b>
MOV_S.NE	b, limm	01110 <b>bbb</b> 110 <b>11111</b>
MOV_S	b, u8	11011 <b>bbb</b> uuuuuuuu
MOV_S	g, h	01000 <b>ggg</b> hhh <b>GG0HH</b>
MOV_S	g, limm	01000 <b>ggg</b> 110 <b>GG011</b>
MOV_S	0, h	01000 <b>110</b> hhh <b>110HH</b>
MOV_S	0, limm	01000 <b>110</b> 11011011011

## MPY, MPY\_S

### Function

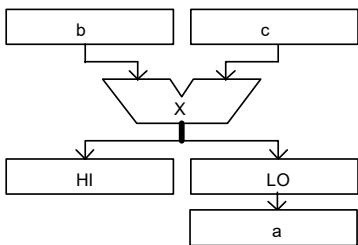
32 x 32 Signed Multiply, returning least-significant 32-bit word of the result.

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
MPY<.f>	a,b,c	00100bbb00011010FBBCCCCCAAAAAA
MPY<.f>	a,b,u6	00100bbb01011010FBBCuuuuuuAAAAAA
MPY<.f>	b,b,s12	00100bbb10011010FBBCssssssSSSSSS
MPY<.cc><.f>	b,b,c	00100bbb11011010FBBCCCCCC0QQQQQ
MPY<.cc><.f>	b,b,u6	00100bbb11011010FBBCuuuuuu1QQQQQ
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:

```
ADD.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](#).

## MPYD

This section describes the standard MPYD instruction.

### Function

Signed 32x32 multiplication. Return a 64-bit result.

### Extension Group

MPY\_OPTION > 7

### Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```



**Note** A is a register pair representing a 64-bit operand, and should be specified as an even numbered register. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### 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

Compute the 32x32 product of the first and second source operands. Assign the 64-bit product to the destination register pair, and then assign the 64-bit product to the 64-bit accumulator (ACCH, ACCL).

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is always cleared, and the sign (N) flag is set to the sign of the result.

## Pseudo Code

```
if(cc) {                                     /* MPYD */
    result = (b * c);
    A = result;
    acc = result;
}
```

## Assembly Code Example

```
MPYD r0,r2,r3      ; 32x32 multiplication of r2 and r3. The result is
; stored in (r1, r0).
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

MPYD<.f>	a, b, c	00101bbb00011000FBBBCCCCCAAAAAA
MPYD<.f>	a, b, u6	00101bbb01011000FBBBuuuuuuAAAAAA
MPYD<.f>	b, b, s12	00101bbb10011000FBBBssssssSSSSSS
MPYD<.cc><.f>	b, b, c	00101bbb11011000FBBBCCCCCC0QQQQQ
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:

```
ADD.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

### Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```



**Note** A is a register pair representing a 64-bit operand, and should be specified as an even numbered register. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### 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

Compute the unsigned 32x32 product of the first and second source operands. Assign the unsigned 64-bit product to the destination register pair, and then assign the 64-bit product to the 64-bit accumulator (ACCH, ACCL).

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is always cleared.

## Pseudo Code

```

if(cc) {
    result = (b * c);
    A = result;
    acc = result;
}
/* MPYDU */

```

## Assembly Code Example

```

MPYDU r0,r2,r3      ; 32x32 unsigned multiplication of r2 and r3. The
;result is stored in (r1,r0)

```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#). For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

MPYDU<.f>	a,b,c	00101bbb00011001FBBBCCCCCCCCAAAAAA
MPYDU<.f>	a,b,u6	00101bbb01011001FBBBuuuuuuAAAAAA
MPYDU<.f>	b,b,s12	00101bbb10011001FBBBssssssSSSSSS
MPYDU<.cc><.f>	b,b,c	00101bbb11011001FBBBCCCCCC0QQQQQ
MPYDU<.cc><.f>	b,b,u6	00101bbb11011001FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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.



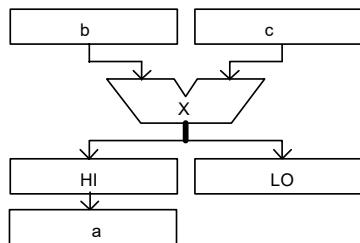
The MPYH mnemonic is deprecated.

### 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 if 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
MPYM<.f>	a,b,c	00100 <b>bbb00011011FBBB</b> CCCCCCCCAAAAAA
MPYM<.f>	a,b,u6	00100 <b>bbb01011011FBBB</b> uuuuuuAAAAAA
MPYM<.f>	b,b,s12	00100 <b>bbb10011011FBBB</b> ssssssssssss
MPYM<.cc><.f>	b,b,c	00100 <b>bbb11011011FBBB</b> CCCCCC0QQQQQ
MPYM<.cc><.f>	b,b,u6	00100 <b>bbb11011011FBBB</b> uuuuuu1QQQQQ

## 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:

```
ADD.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.



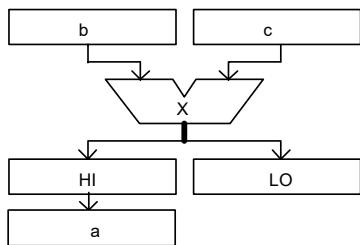
The MPYHU mnemonic is deprecated.

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
MPYMU<.f>	a,b,c	00100bbb00011100FBBBCCCCCCCCAAAAAA
MPYMU<.f>	a,b,u6	00100bbb01011100FBBBuuuuuuAAAAAA
MPYMU<.f>	b,b,s12	00100bbb10011100FBBBssssssSSSSSS
MPYMU<.cc><.f>	b,b,c	00100bbb11011100FBBBCCCCCCC0QQQQQ
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:

```
ADD.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

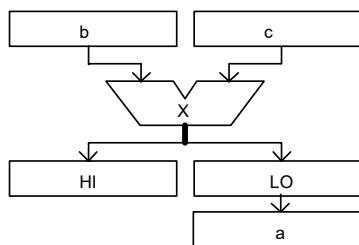
32 x 32 Unsigned Multiply, returning least-significant 32-bit word of result.

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
MPYU<.f>	a,b,c	00100bbb00011101FBBBCCCCCCCCAAAAAA
MPYU<.f>	a,b,u6	00100bbb01011101FBBBuuuuuuAAAAAA
MPYU<.f>	b,b,s12	00100bbb10011101FBBBssssssSSSSSS
MPYU<.cc><.f>	b,b,c	00100bbb11011101FBBBCCCCCCC0QQQQQ
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:

```
ADD.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

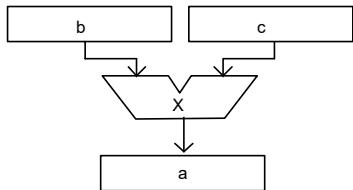
16 x 16 unsigned Multiply, return 32-bit word result.

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
MPYUW<.f>	a,b,c	00100bbb00011111FBBBCCCCCCCCAAAAAA
MPYUW<.f>	a,b,u6	00100bbb01011111FBBBuuuuuuAAAAAA
MPYUW<.f>	b,b,s12	00100bbb10011111FBBBssssssSSSSSS
MPYUW<.cc><.f>	b,b,c	00100bbb11011111FBBBCCCCCCCC0QQQQQ
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:

ADD.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

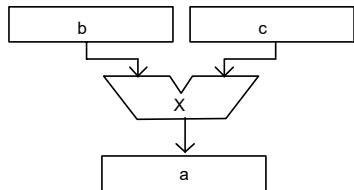
16 x 16 Signed Multiply.

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
MPYW<.f>	a, b, c	00100bbb00011110FBBBCCCCCCCCAAAAAA
MPYW<.f>	a, b, u6	00100bbb01011110FBBBuuuuuuAAAAAA
MPYW<.f>	b, b, s12	00100bbb10011110FBBBssssssSSSSSS
MPYW<.cc><.f>	b, b, c	00100bbb11011110FBBBCCCCCCC0QQQQQ
MPYW<.cc><.f>	b, b, u6	00100bbb11011110FBBBuuuuuu1QQQQQ
MPYW_S	b, b, c	01111bbbcccc01001

## 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:

ADD.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

```
dest = 0 - src /* NEG */
```

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### 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

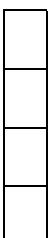
### Instruction Format

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 319](#). For more information about the encodings, see

[“Key for 32-bit Addressing Modes and Encoding Conventions” on page 258](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions” on page 259](#).

#### Instruction Code

NOP_S	0111100011100000
NOP	00100110010010100111000000000000

## NORM

### Function

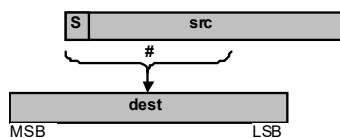
Normalize

### 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 r2write result into r1
```

## Syntax and Encoding

For more information about the encodings, see “Key for 32-bit Addressing Modes and Encoding Conventions” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

NORM<.f>	b, c	00101bbb00101111FBBCCCCCC000001
NORM<.f>	b, u6	00101bbb01101111FBBCuuuuuu000001

## NORMH NORMW

### Function

Normalize 16-bit Half-word



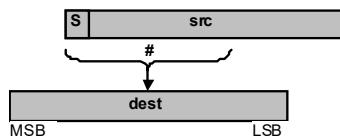
The NORMW mnemonic is deprecated.

### 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++)
    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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

NORMH<.f> b,c	00101bbb00101111FBBBCCCCCC001000
---------------	----------------------------------

NORMH<.f> b,u6 00101bbb01101111FBBBuuuuuuu001000

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

NOT<.f>	b, c	00100bbb00101111FBBBBCCCCC001010
NOT<.f>	b, u6	00100bbb01101111FBBBuuuuuu001010
NOT_S	b, c	01111bbbccc10010

# OR

## Function

Bitwise OR

## Extension Group

BASELINE

## Operation

`if (cc) a = b | 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
OR<.f>	a,b,c	00100bbb00000101FBBBCCCCCAAAAAA
OR<.f>	a,b,u6	00100bbb01000101FBBBuuuuuuAAAAAA
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:

```
ADD.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](#) [LDD](#) [LDL](#)).

### 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 */
SP    = SP + 4
```

## Assembly Code Example

```
POP 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

POP_S	b	11000 <b>bbb</b> 11000001
POP_S	BLINK	11000 <b>RRR</b> 11010001

## PREALLOC

### Function

Allocates a cache line as a preparation for writing the whole line without the overhead of fetching the line from memory.

### Extension Group

OS\_OPT\_OPTION == 1; This option cannot be configured in the ARC EM processor, and hence this instruction is decoded as an illegal instruction in the ARC EM processor.

### Operation

Fetch\_with\_modified\_ownership(L2/external\_memory) AND Fill\_cacheLine\_with\_zeros

### Instruction Format

op b,c

### Syntax Example

PREALLOC<.aa> [b,c]

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

A PREALLOC instruction that misses in the data cache allocates the line in modified state. The line is allocated without the overhead of fetching it from memory. The line is filled with zeros.

A PREALLOC instruction that hits in the data cache or targets any other memory such as peripheral, ICCM, DCCM, or uncached behaves as a NOP.

The PREALLOC instruction does not raise data memory address-related exceptions such as: page fault, memory error, or misaligned exception.



Software must not assume the PREALLOC instruction zeroes out the line, that is the software should explicitly write to the line with store instructions.

## Assembly Code Example

```
PREALLOC r1, r2 ; allocates a cache line as a preparation for  
; writing the whole line without the overhead  
; of fetching the line from memory
```

## Syntax and Encoding

See [Table 8-4](#) on page [721](#) and [Table 8-5](#) on page [722](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
PREALLOC<.aa>	[b, c]	00100bbbbaa110RR10BBBBCCCCCC111110
PREALLOC<.aa>	[b, s9]	00010bbbsssssssssSBBB0aaRR1111110

## 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 5-10](#) on page 262 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 8-4    PREFETCH Instruction Encodings**

Class	<d>	<x>	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 <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 8-5](#). These combinations cover all load and prefetch-type instructions in 32-bit encodings.

**Table 8-5    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 word from memory at address r1+4
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

PREFETCH<.aa>	[b, c]	00100bbbbaa110RR00BBBBCCCCCC111110
PREFETCH<.aa>	[b, s9]	00010bbbssssssssSBBB0aaRR0111110

## PREFETCHW

### Function

Prefetch line from the memory with an intention to write. Move data from the L2 cache or external memory closer to the processor in anticipation of a write

### Extension Group

OS\_OPT\_OPTION == 1. This option cannot be configured in the ARC EM processor, and hence this instruction is decoded as an illegal instruction in the ARC EM processor.

### Operation

Fetch\_with\_exclusive\_ownership(L2/external\_memory)

### Instruction Format

op b,c

### Syntax Example

PREFETCHW<.aa> [b,c]

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

The PREFETCHW instruction prefetches lines from the memory with an intention to write:

- The coherency manager invalidates the line in other cores, getting the line in modified state. Even modified copies are discarded
- This instruction has the same semantics of PREFETCH in a single core system
- There is no requirement for cache-line alignment of prefetch addresses

The PREFETCHW instruction encoding is similar to the PREFETCH instruction except for the .di bits. For PREFETCHW, .di==1.

### Assembly Code Example

```
PREFETCHW [r1, 4]
```

## Syntax and Encoding

See [Table 8-4](#) on page [721](#) and [Table 8-5](#) on page [722](#). For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

<b>Instruction Code</b>		
PREFETCHW<.aa>	[b, c]	00100bbbbaa110RR01BBBBCCCCCC111110
PREFETCHW<.aa>	[b, s9]	00010bbbsssssssssSBBB1aaRR0111110

## PUSH

### Function

Synonym for [ST](#) [STH](#) [STB](#) [STD](#).

### 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 STD](#) 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

PUSH_S	b	11000 <b>bbb</b> 11100001
PUSH_S	blink	11000 <b>RRR</b> 1110001

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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.



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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

RCMP	b,c	00100bbb000011011BBBCCCCCRRRRRR
RCMP	b,u6	00100bbb010011011BBBuuuuuuRRRRRR
RCMP	b,s12	00100bbb100011011BBBssssssSSSSSS
RCMP<.cc>	b,c	00100bbb110011011BBBCCCCC0QQQQQ
RCMP<.cc>	b,u6	00100bbb110011011BBBuuuuuu1QQQQQ

## REM

### Function

2's complement integer Remainder.

### Extension Group

DIV\_Rem\_OPTION

### 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, please refer to [Appendix Table C-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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
REM<.f>	a,b,c	00101bbb00001000FBBCCCCCAAAAAA
REM<.f>	a,b,u6	00101bbb01001000FBBCuuuuuuAAAAAA
REM<.f>	b,b,s12	00101bbb10001000FBBCssssssSSSSSS
REM<.cc><.f>	b,b,c	00101bbb11001000FBBCCCCCC0QQQQQ
REM<.cc><.f>	b,b,u6	00101bbb11001000FBBCuuuuuu1QQQQQ

## 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:

```
ADD.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.

### Extension Group

DIV\_Rem\_Option

### 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 is 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, please refer to [Appendix Table C-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) {                                     /* REMU */
    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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

Instruction Code		
REMU<.f>	a,b,c	00101bbb0001001FBBCCCCCCAAAAAA
REMU<.f>	a,b,u6	00101bbb01001001FBBCuuuuuuAAAAAA
REMU<.f>	b,b,s12	00101bbb10001001FBBCssssssSSSSSS
REMU<.cc><.f>	b,b,c	00101bbb11001001FBBCCCCCC0QQQQQ
REMU<.cc><.f>	b,b,u6	00101bbb11001001FBBCuuuuuu1QQQQQ

## 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:

```
ADD.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

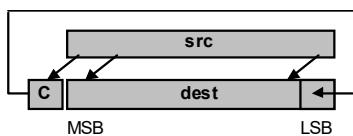
### Extension Group

BASELINE

### Operation

$b = c << 1; b = b | C; C = c >> 31;$

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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

RLC<.f>	b, c	00100bbb00101111FBBBCCCCCC001011
RLC<.f>	b, u6	00100bbb01101111FBBBuuuuuu001011

## ROL

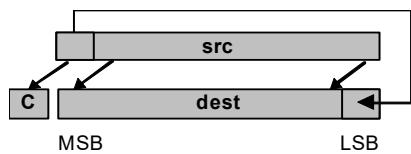
### Function

Rotate Left

### Extension Group

BASELINE

### Operation

$$b = (c << 1) \mid (c >> 31);$$


### Instruction Format

op b,c

### Syntax Example

ROL<.f> b,c

### STATUS32 Flags Affected

Z	<input checked="" type="checkbox"/>	= Set if result is zero
N	<input checked="" type="checkbox"/>	= Set if most-significant bit of result is set
C	<input checked="" type="checkbox"/>	= Set to most-significant bit of source operand
V	<input type="checkbox"/>	= 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

ROL<.f> b,c 00100bbb00101111FBBBBCCCCC001101

ROL<.f> b,u6 00100bbb01101111FBBBuuuuuu001101

## ROL8

### Function

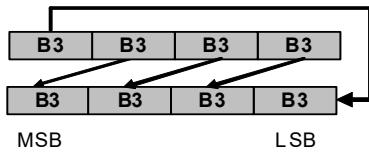
Rotate Left 8

### Extension Group

SHIFT\_OPTION 1 or 3

### Operation

$b = (c << 8) | (c >> 24);$



### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

ROL8<.f> b,c 00101bbb00101111FBBBBCCCCC010000

ROL8<.f> b,u6 00101bbb01101111FBBBuuuuuu010000

## ROR

### Function

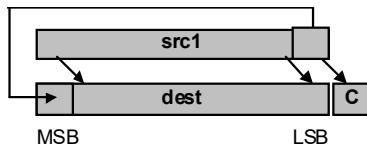
Rotate Right

### Extension Group

BASELINE

### Operation

$C = c \& 1; b = c >> 1 \mid c << 31;$



### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

ROR<.f>	b, c	00100bbb00101111FBBBBCCCCC000011
ROR<.f>	b, u6	00100bbb01101111FBBBuuuuuu000011

## ROR multiple

### Function

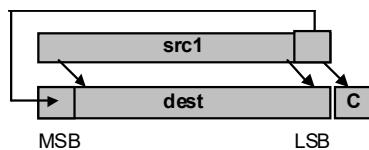
Multiple Rotate Right

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

ROR<.f>	a,b,c	00101bbb00000011FBBBBCCCCCAAAAAAA
ROR<.f>	a,b,u6	00101bbb01000011FBBBBuuuuuuuAAAAAA
ROR<.f>	b,b,s12	00101bbb10000011FBBBssssssSSSSSS
ROR<.cc><.f>	b,b,c	00101bbb11000011FBBBCCCCCC0QQQQQ
ROR<.cc><.f>	b,b,u6	00101bbb11000011FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

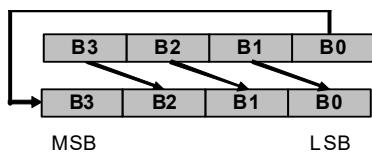
Rotate right 8

### Extension Group

SHIFT\_OPTION 1 or 3

### Operation

$b = (c >> 8) | (c << 24);$



### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

ROR8<.f> b,c 00101bbb00101111FBBBBCCCCC010001

ROR8<.f> b,u6 00101bbb01101111FBBBBuuuuuu010001

## RRC

### Function

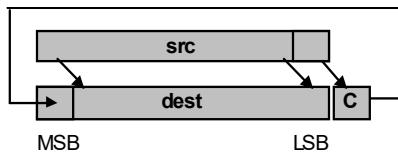
Rotate Right through Carry

### Extension Group

BASELINE

### Operation

$b=c>>1 \mid C << 31; C=c\& 1;$



### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

RRC<.f>	b, c	00100bbb00101111FBBBCCCCCC000100
RRC<.f>	b, u6	00100bbb01101111FBBBuuuuuuu000100

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
RSUB<.f>	a,b,c	00100bbb00001110FBBBCCCCC <del>AAAAAA</del>
RSUB<.f>	a,b,u6	00100bbb01001110FBBBuuuuuu <del>AAAAAA</del>
RSUB<.f>	b,b,s12	00100bbb10001110FBBBssssss <del>SSSSSS</del>
RSUB<.cc><.f>	b,b,c	00100bbb11001110FBBBCCCCC <del>0QQQQQ</del>
RSUB<.cc><.f>	b,b,u6	00100bbb11001110FBBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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]) && (FIRQ_OPTION))
        if (IRQ_PRIORITY_PENDING[0])
            ServiceAnotherFastInterrupt()
        else
            ReturnFromFastInterrupt()
    else
        if (PendingNewIntrPreemptsReturnLevel)
            ServiceAnotherInterrupt()
        else
            ReturnFromInterrupt()
```



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 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 epilogue 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 epilogue sequence is designed to be re-startable such that the stack pointer is not modified until an epilogue 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 handles, 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)                                //see example 4.5
    else
        ReturnFromInterrupt(PI, W)                                    //see example 4.7

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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
SBC<.f>	a,b,c	00100bbb00000011FBBBCCCCCAAAAAAA
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:

```
ADD.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](#).

## SCOND

### Function

Store Conditional

### Extension Group

ATOMIC\_OPTION == 1

### Operation

\_scond (b,c);

### Instruction Format

op b,c

### Syntax Example

SCOND<.di> b, [c]

### STATUS32 Flags Affected

Z		= Set to the value of Lock Flag immediately before the SCOND takes place
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

If the Lock Flag register (LF) is set to 1, the 32-bit value from the b source operand register is written to the memory address given by c source operand register. If LF is set to 0, no memory write takes place.

When this instruction completes, whether it writes to memory or not, a copy of the LF register is assigned to the Z status flag and the LF register is cleared.

This instruction always performs the test of LF and the conditional write to memory as an atomic operation. If the operand location has been written, between the point at which LF is set to 1 by LLOCK, and the execution of the following SCOND, the SCOND does not write to memory and clears the Z flag. As a result, if two processors use SCOND instructions to modify the same address, and both have their respective LF registers set to 1, only one performs the store operation. The value returned to each respective processor's Z flag indicates which processor succeeded in writing, and the value 1 is returned to only one such contending processor.

Each processor is expected to test the Z status result, returned by an SCOND instruction, to see if the LLOCK/SCOND pair succeeded. Software typically repeats the LLOCK/SCOND sequence until success is detected. The success or failure of the SCOND instruction is returned to the processor via the Zero

flag (STATUS32[Z]), which is set to 1 if the SCOND instruction succeeded in writing to memory, and is cleared if the write to memory did not take place.

The SCOND.DI instruction operates directly on external memory, bypassing any data cache that may be present in the path from the processor to the physical memory system. This form must be used in implementations of the ARCv2 that do not provide hardware cache coherency for shared memory accesses.

The memory write operations implied by the SCOND instruction, without the .DI modifier, are cached writes. This form of the instruction must be used only in implementations of the ARCv2 that provide hardware cache coherency for shared memory accesses, or in systems containing only a single processor.

The effective address of SCOND must be aligned to a 4-byte boundary, otherwise an EV\_Misaligned exception is raised even if STATUS32.AD==1.

## Pseudo Code

```
if (LF == 1)
{
    EA = PhysicalAddress (c)
    WriteWord(EA[31:2], b)      ; for non-PAE builds; PAE builds use EA[39:2]
}

STATUS32.Z = LF
LF = 0
```

## Assembly Code Example

```
SCOND    r1, [r2]    ; Store r1 to address given by r2 if LF == 1
           ; Set STATUS32.Z to LF and then clear LF
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

SCOND<.di> b, [c]	00100 <b>bbb</b> 00101111D <b>BBB</b> CCCCCC010001
SCOND<.di> b, [u6]	00100 <b>bbb</b> 01101111D <b>BBB</b> uuuuuu010001

## SCONDD

### Function

Store conditional on 64-bit data

### Extension Group

(ATOMIC\_OPTION == 1) and (LL64\_OPTION==1)

### Operation

\_scond (b,c);

### Instruction Format

op b,c

### Syntax Example

SCONDD<.di> b, [c]

### STATUS32 Flags Affected

Z	•	= Set to the value of the Lock Flag immediately before the instruction is executed
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

The SCONDD instruction is similar to the [SCOND](#) instruction except that SCONDD operates on 64-bit data objects and (if successful) it stores a pair of 32-bit core registers to the destination memory address.

The effective address of SCONDD must be aligned to an eight-byte boundary. Otherwise an [EV\\_Misaligned](#) exception is raised even if STATUS32 .AD==1.

## Pseudo Code

```
if (LF == 1)
{
    EA = PhysicalAddress (c)
    WriteDoubleWord(EA[31:3], b); for non-PAE builds; PAE builds use [39:3]
}

STATUS32.Z = LF
LF = 0
```

## Assembly Code Example

```
SCONDD r0, [r2] ; Store register pair(r0,r1) to address given by r2
                  ; if LF == 1, set STATUS32.Z to LF and then clear LF
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

SCONDD<.di> b, [c] 00100bbb00101111DBBBCCCCC010011

SCONDD<.di> b, [u6] 00100bbb01101111DBBBuuuuuu010011

## 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 $(\text{src1} - \text{src2})$ is negative
C		= Set if carry is generated by $(\text{src1} - \text{src2})$
V		= Set if an overflow is generated by $(\text{src1} - \text{src2})$

### 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
dest = (src1 Relop src2)                                     /* SETCC */
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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
SETcc<.f>	a,b,c	00100bbb00iiiiifBBBCCCCCAAAAAAA
SETcc<.f>	a,b,u6	00100bbb01iiiiifBBBuuuuuuAAAAAAAA
SETcc<.f>	b,b,s12	00100bbb10iiiiifBBBssssssSSSSSS
SETcc<.cc><.f>	b,b,c	00100bbb11iiiiifBBBCCCCC0QQQQQ
SETcc<.cc><.f>	b,b,u6	00100bbb11iiiiifBBBuuuuuu1QQQQQ

## 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:

```
ADD.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 encodings

Sub-opcode	iiiiii	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

## 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 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

```

{
    /* SETI */

    if (src2[5] == 1)
        STATUS32.E <- src2[3:0]
        STATUS32.IE <- src2[4]
    else
        STATUS32.IE <- 1
        if (src2[4] == 1)
            STATUS32.E <- src2[3:0]
}

if (SEC_MODES_OPTION == 1)          /*SETI*/
/*SecureShield is present*/
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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
SETI	c	00100110001011110000CCCCCC111111
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 in order to provide a concise way of specifying “enable interrupts”.

SETI is not a serializing instruction. SETI therefore takes a single cycle unless stalled due to ordinary register operand dependencies.

## SEXB

### Function

Sign Extend Byte

### Extension Group

BASELINE

### Operation

$b = (c << 24) >> 24;$

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

SEXB<.f>	b,c	00100bbb00101111FBBBBCCCCC000101
SEXB<.f>	b,u6	00100bbb01101111FBBBBuuuuuu000101
SEXB_S	b,c	01111bbbccc01101

## SEXH SEXW

### Function

Sign Extend Half-word 16-bits to word



The SEXW mnemonic is deprecated.

### Extension Group

BASELINE

### Operation

$b = (c << 16) >> 16;$

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

SEXH<.f>	b,c	00100 <b>bbb</b> 00101111 <b>F</b> BBBCCCCC <b>000110</b>
SEXH<.f>	b,u6	00100 <b>bbb</b> 01101111 <b>F</b> BBBuuuuuu <b>000110</b>
SEXH_S	b,c	01111 <b>bbbccc</b> 01110

## 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

SFLAG	c	00110000001011110000CCCCCC111111
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 <a href="#">Table 8-6</a>
E[3:0]	•	Updated depending on SLEEP operand as explained in <a href="#">Table 8-6</a>
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
ZZ	•	= 1
SM	•	Updated depending on SLEEP operand as explained in <a href="#">Table 8-6</a>

### 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 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 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 8-2 SLEEP Operand**



**Table 8-6 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 8-5 Sleep placement in code

```

SUB r2, r2, 0x1
ADD r1, r1, 0x2
SLEEP
...

```

**Example 8-6** illustrates the use of the SLEEP instruction with each combination of interrupts levels enabled or disabled during sleep.

### Example 8-6 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

SLEEP	c	00100001001011110000CCCCCC111111
SLEEP	<u6>	00100001011011110000uuuuuu111111

## 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.

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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

```
SJLI u12 00101RRR101000001RRRuuuuuuuUUUUUUU
```

## 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 Aux. register pointed to by r2
```

### Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

**Instruction Code**

SR b, [c]

SR b, [u6] 00100bbb01101011RBBBuuuuuuRRRRRR

SR b, [s12] 00100bbb10101011RBBBssssssSSSSSS

## ST STH STB STD

### Function

Store to Memory

### Extension Group

ST, STH, STB: BASELINE

STD: LL64\_OPTION

ST\_S R0,[GP,s11]: CODE\_DENSITY

### 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 5-13 on page 263 lists the data size modes.



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 5-12](#) on page 263 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 5-10](#) on page 262 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).

A Store Double (STD) instruction that specifies a  $R_n$  register as the source of the store data combines 32 bits of data from the lower 32 bits of registers  $R_n$  and  $R_{n+1}$  to form the 64-bit store data value. The effective address given in the instruction is used to store a data word from the contents of the even register  $R_n$ , and the address +4 is used to store the contents of the odd register  $R_{n+1}$ . If  $R_n$  is an odd-numbered register, an [Illegal Instruction](#) exception is raised.

An STD instruction that specifies either a Long Immediate data value (limm) or a signed 6-bit short constant (w6) as the store data, sign extends that value to 64 bits before storing it.

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

<b>Instruction Code</b>		
ST<zz><.aa><.di>	c, [b]	00011bbb000000000BBBCCCCC0DaaZZ0
ST<zz><.aa><.di>	c, [b, s9]	00011bbbsssssssssSBBBCCCCC0DaaZZ0
ST<zz><.aa><.di>	w6, [b]	00011bbb000000000BBBwwwwwwDaaZZ1
ST<zz><.aa><.di>	w6, [b, s9]	00011bbbsssssssssSBBBwwwwwwDaaZZ1
ST_S	b, [SP, u7]	11000bbb010uuuuuu
ST_S	c, [b, u7]	10100bbbcccuuuuuu
STB_S	b, [SP, u7]	11000bbb011uuuuuu



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]	10101bbbcccuuuuuu
STH_S	c, [b, u6]	10110bbbcccuuuuuu

Encodings supported by LL64\_OPTION

<b>Instruction Code</b>		
STD<.aa><.di>	c, [b]	00011bbb000000000BBBCCCCC0Daa110
STD<.aa><.di>	c, [b, s9]	00011bbbsssssssssSBBBCCCCC0Daa110
STD<.aa><.di>	w6, [b]	00011bbb000000000BBBwwwwwwDaa111
STD<.aa><.di>	w6, [b, s9]	00011bbbsssssssssSBBBwwwwwwDaa111

## 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:

```
ADD.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.



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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

SUB<.f>	a,b,c	00100bbb00000010FBBBCCCCCAAAAAA
SUB<.f>	a,b,u6	00100bbb01000010FBBBuuuuuuAAAAAA
SUB<.f>	b,b,s12	00100bbb10000010FBBBssssssSSSSSS
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	01111bbb11000000
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:

```
ADD.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 << 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
SUB1<.f>	a,b,c	00100bbb00010111FBBBCCCCCCCCAAAAAA
SUB1<.f>	a,b,u6	00100bbb01010111FBBBuuuuuuAAAAAA
SUB1<.f>	b,b,s12	00100bbb10010111FBBBssssssssssss
SUB1<.cc><.f>	b,b,c	00100bbb11010111FBBBCCCCCCC0QQQQQ
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:

```
ADD.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 << 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
SUB2<.f>	a,b,c	00100bbb00011000FBBBCCCCCCCCAAAAAA
SUB2<.f>	a,b,u6	00100bbb01011000FBBBuuuuuuAAAAAA
SUB2<.f>	b,b,s12	00100bbb10011000FBBBssssssssssss
SUB2<.cc><.f>	b,b,c	00100bbb11011000FBBBCCCCCCC0QQQQQ
SUB2<.cc><.f>	b,b,u6	00100bbb11011000FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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 << 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
SUB3<.f>	a,b,c	00100bbb00011001FBBBCCCCCCCCAAAAAA
SUB3<.f>	a,b,u6	00100bbb01011001FBBBuuuuuuAAAAAA
SUB3<.f>	b,b,s12	00100bbb10011001FBBBssssssssssss
SUB3<.cc><.f>	b,b,c	00100bbb11011001FBBBCCCCCCC0QQQQQ
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:

```
ADD.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

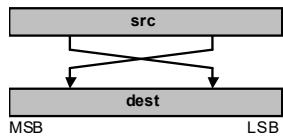
Swap 16-bits half-words

### 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

SWAP<.f> b, c 00101bbb00101111FBBCCCCCC000000

SWAP<.f> b, u6 00101bbb01101111FBBCuuuuuu000000

## SWAPE

### Function

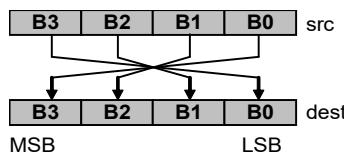
Swap byte ordering

### 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	<input checked="" type="checkbox"/>	= Set if result is zero
N	<input checked="" type="checkbox"/>	= Set if most-significant bit of result is set
C	<input type="checkbox"/>	= Unchanged
V	<input type="checkbox"/>	= 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

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 hence 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



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. If the Exception Fault Address register (see [Exception Fault Address, EFA](#)) is present, this register 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

**Instruction Code**

SWI            0010001001101110000RRRRRR111111

SWI\_S          0111101011100000

SWI\_S n6\*     01111nnnnnn11111

\*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.



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 – hence a bogus interrupt is generated.

In order 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### 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 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 : 0x26 : 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

TRAP_S u6	01111 <u>uuuuuu</u> 11110
-----------	---------------------------

## 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.



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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
TST	b, c	00100 <b>bbb</b> 00010111 <b>BBB</b> CCCCC <b>RRRRR</b> R
TST	b, u6	00100 <b>bbb</b> 010010111 <b>BBB</b> uuuuuu <b>RRRRR</b> R
TST	b, s12	00100 <b>bbb</b> 100010111 <b>BBB</b> ssssss <b>SSSSS</b> S
TST<.cc>	b, c	00100 <b>bbb</b> 110010111 <b>BBB</b> CCCCC <b>QQQQQ</b> Q
TST<.cc>	b, u6	00100 <b>bbb</b> 110010111 <b>BBB</b> uuuuuu <b>1QQQQ</b> Q
TST_S	b, c	01111 <b>bbbccc</b> 01011

## 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:

```
ADD.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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

UNIMP_S	0111100111100000
---------	------------------

## 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:

```
ADD.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

(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 the vector operands, see “[Data Formats](#)” on page [80](#).

### 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 LP\_COUNT (r60) is given as the destination register.

## Pseudo Code

```
if (cc)== true then                                /* VADD2H */
    a.h0 = b.h0+c.h0
    a.h1 = b.h1+c.h1
```

## 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](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

VADD2H	a,b,c	00101bbb000101000BBBBCCCCCCCCAAAAAA
VADD2H	a,b,u6	00101bbb010101000BBBBuuuuuuuuAAAAAA
VADD2H	b,b,s12	00101bbb100101000BBBBssssssSSSSSS
VADD2H<.cc>	b,b,c	00101bbb110101000BBBBCCCCCCC0QQQQQ
VADD2H<.cc>	b,b,u6	00101bbb110101000BBBBuuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## VADDSSUB2H

### 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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VADDSSUB2H 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 are 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) {                                     /* VADDSSUB2H */
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```

## Assembly Code Example

```
VADDSSUB2H r1,r2,r3      ; dual SIMD 16-bit addition and subtraction
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

VADDSSUB2H	a, b, c	00101bbb000101100 BBBCCCCCAAAAAA
VADDSSUB2H	a, b, u6	00101bbb010101100 BBBuuuuuuAAAAAA
VADDSSUB2H	b, b, s12	00101bbb100101100 BBBssssssSSSSSS
VADDSSUB2H<.cc>	b, b, c	00101bbb110101100 BBBCCCCC0QQQQQ
VADDSSUB2H<.cc>	b, b, u6	00101bbb110101100 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## 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:

```
ADD .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

This section describes the standard MPYD instruction.

### 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

The 32-bit b and c operands specify vectors containing two 16-bit signed elements. Each pair of elements from b and c is multiplied to form two 32-bit signed products. These products are added to the accumulator, and the resulting value is assigned to the destination register if defined. The destination operand is a register pair, A, and should be an even-numbered register. This instruction does not modify any flags.

## Pseudo Code

```

if (cc) {                                     /* VMAC2H*/
    A.w0 = (acc.w0 += (b.h0 * c.h0));
    A.w1 = (acc.w1 += (b.h1 * c.h1));
}

```

## 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

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
VMAC2H	a,b,c	00101bbb000111100 BBBCCCCCAAAAAA
VMAC2H	a,b,u6	00101bbb010111100 BBBuuuuuuuAAAAAA
VMAC2H	b,b,s12	00101bbb100111100 BBBssssssSSSSSS
VMAC2H<.cc>	b,b,c	00101bbb110111100 BBBCCCCC0QQQQQ
VMAC2H<.cc>	b,b,u6	00101bbb110111100 BBBuuuuuuu1QQQQQ

## 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:

```
ADD.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

This section describes the standard MPYD instruction.

### 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

The 32-bit b and c operands specify vectors containing two 16-bit unsigned elements. Each pair of elements from b and c is multiplied to form two 32-bit unsigned products. These products are added to the accumulator, and the resulting value is assigned to the destination register if defined. The destination operand is a register pair, A, and should be an even-numbered register. This instruction does not modify any flags.

## Pseudo Code

```

if (cc) {                                     /* VMAC2HU */
    A.w0 = (acc.w0 += (b.h0 * c.h0));
    A.w1 = (acc.w1 += (b.h1 * c.h1));
}

```

## 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

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

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:

```
ADD.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

This section describes the standard MPYD instruction.

### 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;
}
```



#### Note

w0 (lower) and w1 (upper) represent the 32-bit elements of a 64-bit operand. A, B, and C are register pairs and represent 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page 80.

### Instruction Format

op A, b, c

### Syntax Example

VMPY2H 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 signed elements. Each pair of elements from b and c is multiplied to form two 32-bit signed products. These products are assigned to the accumulator, and also to the destination register if defined.

This instruction does not modify any flags.

## Pseudo Code

```
if(cc) {                                     /* VMPY2H */
    acc.w0 = A.w0 = b.h0 * c.h0;
    acc.w1 = A.w1 = b.h1 * c.h1;
}
```

## Assembly Code Example

```
VMPY2H r0,r2,r3 ; dual SIMD 16-bit multiplication
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

VMPY2H	a, b, c	00101bbb000111000 BBBCCCCCAAAAAA
VMPY2H	a, b, u6	00101bbb010111000 BBBuuuuuuAAAAAA
VMPY2H	b, b, s12	00101bbb100111000 BBBssssssSSSSSS
VMPY2H<.cc>	b, b, c	00101bbb110111000 BBBCCCCC0QQQQQ
VMPY2H<.cc>	b, b, u6	00101bbb110111000 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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

This section describes the standard MPYD instruction.

### 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;
}
```



#### Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. `w0` (lower) and `w1` (upper) represent the 32-bit elements of a 64-bit operand. `A` is a register pair and represents a 64-bit operands, and should be specified as an even numbered register. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A,b,c

### Syntax Example

VMPY2HU 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 unsigned elements. Each pair of elements from `b` and `c` is multiplied to form two 32-bit unsigned products. These products are assigned to the accumulator, and also the destination register pair if defined.

This instruction does not modify any flags.

## Pseudo Code

```
if(cc) {                                     /* VMPY2HU */
    acc.w0 = A.w0 = b.h0 * c.h0;
    acc.w1 = A.w1 = b.h1 * c.h1; }
```

## Assembly Code Example

```
VMPY2HU r0,r2,r3 ; dual SIMD 16-bit unsigned multiplication. The
; result is stored in (r1,r0)
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

VMPY2HU	a, b, c	00101bbb000111010 BBBCCCCCAAAAAA
VMPY2HU	a, b, u6	00101bbb010111010 BBBuuuuuuAAAAAA
VMPY2HU	b, b, s12	00101bbb100111010 BBBssssssSSSSSS
VMPY2HU<.cc>	b, b, c	00101bbb110111010 BBBCCCCC0QQQQQ
VMPY2HU<.cc>	b, b, u6	00101bbb110111010 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## 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:

```
ADD .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](#)” on page [80](#).

### 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



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

VSUB2H	a,b,c	00101bbb000101010 BBBCCCCCAAAAAA
VSUB2H	a,b,u6	00101bbb010101010 BBBuuuuuuAAAAAA
VSUB2H	b,b,s12	00101bbb100101010 BBBssssssSSSSSS
VSUB2H<.cc>	b,b,c	00101bbb110101010 BBBCCCCC0QQQQQ
VSUB2H<.cc>	b,b,u6	00101bbb110101010 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## 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:

```
ADD .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](#).

## 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:

```
ADD .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](#)” on page [80](#).

### 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) {                                     /* VSUBADD2H */
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 + c.h1;
}
```

## Assembly Code Example

```
VSUBADD2H r1,r2,r3      ; dual SIMD 16-bit subtraction and addition of r2
                           ; and r3
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code			
VSUBADD2H	a, b, c	00101bbb000101110BBBCCCCC <del>AAAAAA</del>	
VSUBADD2H	a, b, u6	00101bbb010101110BBBuuuuuu <del>AAAAAA</del>	
VSUBADD2H	b, b, s12	00101bbb100101110BBBsssss <del>SSSSSS</del>	
VSUBADD2H<.cc>	b, b, c	00101bbb110101110BBBCCCCC <del>0QQQQQ</del>	
VSUBADD2H<.cc>	b, b, u6	00101bbb110101110BBBuuuuuu <del>1QQQQQ</del>	

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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 <a href="#">Table 8-6</a>
E[3:0]	•	Updated depending on the SLEEP operand as explained in <a href="#">Table 8-6</a>
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
ZZ	•	= 1
SM	•	Updated depending on the SLEEP operand as explained in <a href="#">Table 8-6</a>

### 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](#)” on page [777](#).

## 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 8-7 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

**Instruction Code**

WEVT	c	0010000001011110001CCCCC111111
WEVT	<u6>	0010000001011110001uuuuuu111111

## WLFC

### Function

Enter the sleep state to reduce dynamic power during busy-waiting loops

### Extension Group

ATOMIC\_OPTION == 1. This option cannot be configured in the ARC EM processor, and hence this instruction is decoded as an illegal instruction in the ARC EM processor.

### Operation

DEBUG[ZZ] = 1;

### Instruction Format

op c

### Syntax Example

WLFC

### STATUS32 Flags Affected

IE	•	Updated depending on the SLEEP operand as explained in <a href="#">Table 8-6</a>
E[3:0]	•	Updated depending on the SLEEP operand as explained in <a href="#">Table 8-6</a>
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
ZZ	•	= 1
SM	•	Updated depending on the SLEEP operand as explained in <a href="#">Table 8-6</a>

### Description

The ARCv2-based processor enters the sleep state when the processor encounters the WLFC instruction. The processor stays in the sleep state until one of the following events occurs:

- ❑ The lock flag (LF) is cleared by another core or a DMA device writing to the address contained in the LPA register. This condition applies only when the LLOCK and SCOND instructions are implemented.
- ❑ An interrupt is taken.
- ❑ An external halt request is acknowledged by the processor.

- A debug transaction places the processor in a halt state.

The WLFC instruction is similar to the SLEEP instruction and is available in both kernel and user modes.

- To access the WLFC instruction in user mode, ensure that STATUS32.US == 1.
- If STATUS32.US == 0 and you try to use the WLFC instruction in user mode, the instruction behaves as a NOP.

The instruction has the same operands as the SLEEP instruction. For more information about how the SLEEP instruction operands work, see “[SLEEP instruction operands](#)” on page [777](#).

## Pseudo Code

```
WLFC (operand)                                     /* WLFC */
    if (LF == 1) {
        if ((STATUS32.U == 0) || STATUS32.US ==
1))
            SLEEP operand
    }
```

## Assembly Code Example

The WLFC instruction can be placed anywhere in the instruction sequence.

### Example 8-8 WLFC Placement in Code

```
SUB r2, r2, 0x1
ADD r1, r1, 0x2
WLFC
...
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

WLFC	c	00100001001011110001CCCCCCC111111
WLFC	<u6>	00100001011011110001uuuuuu111111

## 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, LIMM

op a, LIMM, 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 *the* 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]                                /* XBFU */
M = src2[9:5] + 1                            *first bit to extract
if cc==true then                           *number of bits to
    dest = (src1 >> N) && ((1 << M)-1)      *extract
    if F==1 then
        Z_flag = if dest==0 then 1 else 0

```

## 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 thru 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” on page 258 and “Key for 16-bit Addressing Modes and Encoding Conventions” on page 259.

### Instruction Code

XBFU<.f>	a,b,c	00100bbb00101101FBBBCCCCCCCCAAAAAA
XBFU<.f>	a,b,u6	00100bbb01101101FBBBuuuuuuuAAAAAA
XBFU<.f>	b,b,s12	00100bbb10101101FBBBssssssSSSSSS
XBFU<.cc><.f>	b,b,c	00100bbb11101101FBBBCCCCCCC0QQQQQ
XBFU<.cc><.f>	b,b,u6	00100bbb11101101FBBBuuuuuuu1QQQQQ

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 LIMM value as the second source operand, that is, using format a , b , c where c is a LIMM.

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 thru 4 are assumed to be zero. The N operand can be fully-specified as bits 0 thru 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

$\text{if } (\text{cc}) \text{ a} = \text{b} \wedge \text{c};$

## 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](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
XOR<.f>	a,b,c	00100bbb00000111FBBCCCCCAAAAAA
XOR<.f>	a,b,u6	00100bbb01000111FBBUuuuuuAAAAAA
XOR<.f>	b,b,s12	00100bbb10000111FBBSssssssSSSSSS
XOR<.cc><.f>	b,b,c	00100bbb11000111FBBCCCCCC0QQQQQ
XOR<.cc><.f>	b,b,u6	00100bbb11000111FBBUuuuuu1QQQQQ
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:

```
ADD.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](#).



# 9

# The Host

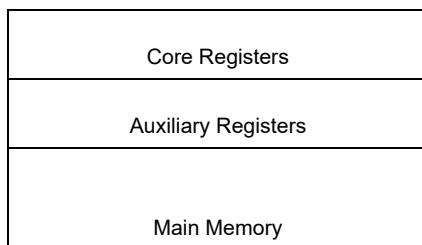
## 9.1 The Host Interface

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 9-1](#) and [Figure 9-2](#) 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 9-1 Example Host Memory Maps, Contiguous Host Memory**



Host I/O Map

**Figure 9-2 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 9-1](#) on page [850](#).

**Table 9-1 Host Accesses to the ARCv2-based processor**

	<b>Running</b>	<b>Halted</b>
<b>Memory</b>	Read/Write	Read/Write
<b>Auxiliary Registers</b>	Read/Write	Read/Write
<b>Core Registers</b>	Read/Write	Read/Write

## 9.2 Core and Auxiliary Registers

The core registers of ARCv2-based processor are available to be read and written by the host. Although these registers can be accessed by the host when an ARCv2-based processor is running, read or write to the core registers from a debug host when the processor is halted.

## 9.3 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.

## 9.4 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.

## 9.5 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.

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.

## 9.6 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 9-2](#) on page 851).

**Table 9-2 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.

### 9.6.1 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 [776](#) and “[Null Instruction Format](#)” on page [93](#) for further details.

### 9.6.2 BRK Instruction in Single Instruction Step Mode

The [BRK](#) instruction behaves exactly as when the processor is not in the Single Step Mode.

## 9.7 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”



# 10

## 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](#)”

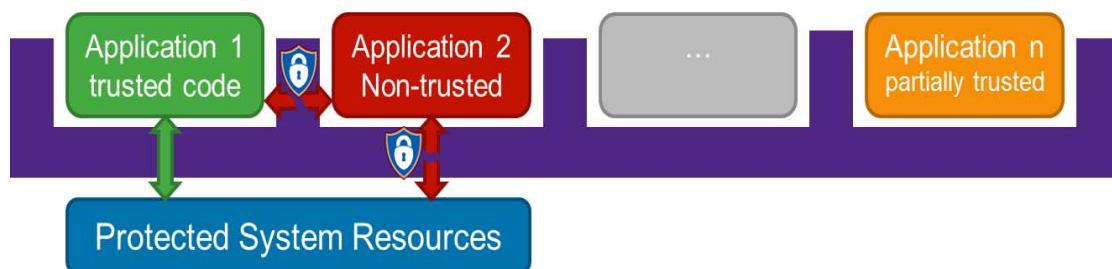


# 11

## 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 11-1 SecureShield Overview**



### 11.1 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

#### 11.1.1 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.

## 11.1.2 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.

### 11.1.2.1 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. 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.

### 11.1.2.2 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.

#### 11.1.2.3 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.

#### 11.1.3 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

##### 11.1.3.1 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 11-1](#).

**Table 11-1 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

### 11.1.3.2 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.

### 11.1.4 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.

### 11.1.5 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 11-2](#) lists the set of additional instructions and auxiliary registers that are available.

**Table 11-2 ARCv2 Instruction Set Options**

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-sec_modes_option	false	-	
	true	SJLI, SFLAG	<ul style="list-style-type: none"> <li>■ NSC_TABLE_TOP</li> <li>■ NSC_TABLE_BASE</li> <li>■ AUX_KERNEL_SP</li> <li>■ AUX_S_USER_SP</li> <li>■ AUX_S_KERNEL_SP</li> <li>■ AUX_SEC_EXCEPT</li> <li>■ ERSEC_STAT</li> <li>■ SEC_STAT</li> <li>■ SEC_VECBASE_BUILD</li> </ul>
-intvbase_preset_s	PC range		<p>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.</p>

# 12

## Secure Pipeline

Secure pipelining supports the following features on the CCMs and caches:

- Scrambled Instruction Execution
- Scrambled Data Access
- Address and Data Scrambling

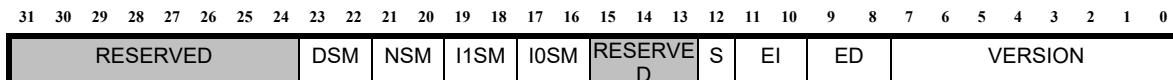
Secure pipelining also supports ECC protection on the register file and program counter. The secure pipeline features are configurable using ARChitect. The corresponding build configuration register is described in this chapter. The ECC protection on the register file and the program counter registers are described in the “[Error Protection](#)” chapter.

## 12.1 Secure Build Configuration Register, SEC\_BUILD

Address: 0xDB

Access: R

**Figure 12-1 SEC\_BUILD Register**



This register indicates the presence of the secure pipeline feature and its configuration, and the CCMs secure and normal memory configurations.

**Table 12-1 SEC\_BUILD Field Description**

Field	Bit	Description
VERSION	[7:0]	<p>Version number</p> <ul style="list-style-type: none"> <li>■ 0x3 : indicates the presence of the following security features:           <ul style="list-style-type: none"> <li>- scrambling and encryption of the cache.</li> <li>- secure pipeline</li> <li>- secure debug</li> <li>- secure MPU</li> <li>- SID protection</li> <li>- key storage protection by MPU</li> <li>- support for the secure kernel and secure user modes</li> <li>- programmable exception handling modes</li> <li>- support for secure interrupts</li> </ul> </li> </ul>
ED	[9:8]	<ul style="list-style-type: none"> <li>■ 00: Data encryption not supported.</li> <li>■ 01: Data encryption is supported; Read-Modify-Write for sub-word writes is controlled by user encryption logic.</li> </ul> <p>All other values are reserved.</p>
EI	[11:10]	<ul style="list-style-type: none"> <li>■ 00: Instruction encryption is not supported.</li> <li>■ 01: 16-bit encryption is supported for instructions in the fetch and DA pipeline stages.</li> </ul> <p>All other values are reserved.</p>

**Table 12-1 SEC\_BUILD Field Description**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
S	[12]	<ul style="list-style-type: none"> <li>■ Indicates that the core contains the secure kernel/user modes and normal kernel/user modes.</li> <li>■ 0x0: includes only normal kernel and user modes.</li> <li>■ 0x1: includes secure kernel/user modes and normal kernel/user modes.</li> </ul> <p>This bit is valid only when -sec_modes_options==true. Otherwise, this bit is read as zero.</p>
I0SM	[17:16]	<p>ICCM0 secure configuration:</p> <ul style="list-style-type: none"> <li>■ 0x0: entire ICCM0 region is normal.</li> <li>■ 0x1: entire ICCM0 is secure region.</li> <li>■ 0x2: Lower half of ICCM0 is normal region; upper-half is secure region.</li> <li>■ 0x3: reserved.</li> </ul> <p>This bit is valid only when -sec_modes_options==true. Otherwise, this bit is read as zero.</p>
I1SM	[19:18]	<p>ICCM0 secure configuration:</p> <ul style="list-style-type: none"> <li>■ 0x0: entire ICCM1 region is normal.</li> <li>■ 0x1: entire ICCM1 is secure region.</li> <li>■ 0x2: Lower half of ICCM1 is normal region; upper-half is secure region.</li> <li>■ 0x3: reserved.</li> </ul> <p>This bit is valid only when -sec_modes_options==true. Otherwise, this bit is read as zero.</p>
NSM	[21:20]	<p>Non-volatile memory secure configuration:</p> <ul style="list-style-type: none"> <li>■ 0x0: entire NV_ICCM region is normal.</li> <li>■ This bit is always set to 0.</li> </ul> <p>This bit is valid only when -sec_modes_options==true. Otherwise, this bit is read as zero.</p>
DSM	[23:22]	<p>ICCM0 secure configuration:</p> <ul style="list-style-type: none"> <li>■ 0x0: entire DCCM region is normal.</li> <li>■ 0x1: entire DCCM is secure region.</li> <li>■ 0x2: Lower half of DCCM is normal region; upper-half is secure region.</li> <li>■ 0x3: reserved.</li> </ul> <p>This bit is valid only when -sec_modes_options==true. Otherwise, this bit is read as zero.</p>



# 13

## Secure Memory Protection Unit

ARC EM 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](#).

### 13.1 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.

### 13.2 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.

### 13.2.1 Enabling the Memory-Protection Unit

The MPU is always enabled on reset.

If the `-mpu_sid_option==true`, the reset value is 0x4001\_81C0, that is:

All accesses are treated as secure: read, write, and execute operations to the default memory region are permitted in the secure user and secure kernel modes. To disable any of the permissions, use the `MPU_EN` register.

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 user and kernel modes.

**Table 13-1** lists the registers and the instructions that are available in the core when you configure the Secure MPU and SID protection.

**Table 13-1 ARCv2 Instruction Set Options**

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
<code>-mpu_sid_option</code>	false	-	
	true	-	<ul style="list-style-type: none"> <li>■ Secure MPU registers</li> <li>■ KEY_STORE_SID</li> </ul>

## 13.3 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_RENDER` register.



- 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.

### 13.3.1 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.

## 13.4 Software Lookup of the MPU Table

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.

To retrieve this information for an address, specify the address in the MPU\_PROBE register.

- If any region hit, the MPU\_INDEX register shows the hit region number. The MPU\_RSTART/MPU\_RENDER registers show the start/end address of this region. The MPU\_RPER register shows the permission configure for this region (including external pins S input).
- If there is multi region hit, then 0x40000000 is readback in the MPU\_INDEX register, the read data of MPU\_RPER/MPU\_RSTART/MPU\_RENDER is set zero.

- If no region matches the address, then 0x80000000 is set in the MPU\_INDEX register. Also, the MPU\_RPER register shows the permission configuration of the default region (same as MPU\_EN). The MPU\_RSTART/REND registers are read as zero.
- 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 define of the region selected by MPU\_INDEX (not including external pins S input), if user write MPU\_PROBE, then MPU\_RPER is updated to show the permission of this address (including external pins S input).

## 13.5 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 MPU region is defined as a normal region, the instruction fetch from this address is normal. 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, exception is not raised.
- 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.

### 13.5.1 Protection Exception Priority

If a S mode violation and an SID violation occur simultaneously, the S mode violation is reported.

## 13.6 MPU Auxiliary Register Interface

**Table 13-2 MPU Registers**

Address	Name	Description
0x6D	“MPU Build Configuration Register, MPU_BUILD”	MPU Build configuration register
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_RENDER”	MPU region end address

**Table 13-2 MPU Registers**

Address	Name	Description
0x44B	"MPU Region Permission Register, MPU_RPER"	MPU region permission register
0x44C	"MPU Probe Register, MPU_PROBE"	MPU Probe register

### 13.6.1 MPU Enable Register, MPU\_EN

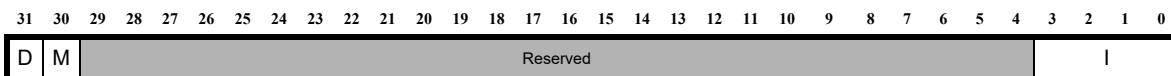
For information about the MPU\_EN register, see [MPU Enable Register, MPU\\_EN](#).

### 13.6.1.1 MPU Index Register, MPU\_INDEX

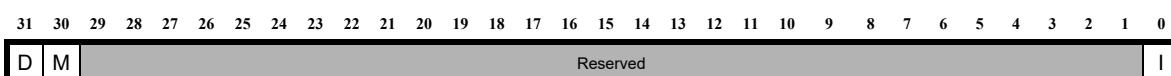
Address: 0x448

Access RW in the secure mode only

**Figure 13-1 MPU\_INDEX for 16 Regions**



**Figure 13-2 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.

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 13-3 MPU\_INDEX Field Description**

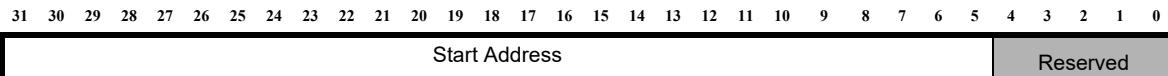
Field	Description
I	Indicates the region index.
M	<p>Indicates that multiple regions include the lookup address specified in the MPU_PROBE register. This bit is read-only.</p> <ul style="list-style-type: none"> <li>■ 0: the lookup address is not present in multiple regions.</li> <li>■ 1: indicates that multiple regions match the lookup address. The <code>MPU_INDEX[I]</code> and <code>MPU_INDEX[D]</code> bits are set to zero. Also, reads of the <code>MPU_RSTART</code>, <code>MPU_RENDER</code>, and <code>MPU_RPER</code> registers return zero.</li> </ul> <p>This bit retains the value of a probe operation until another probe operation is issued.</p>
D	<p>Indicates if the lookup address specified in the MPU_PROBE register matches the default region. This bit is read-only.</p> <ul style="list-style-type: none"> <li>■ 0: the lookup address is not located in the default region.</li> <li>■ 1: the lookup address is located in the default region. The <code>MPU_EN</code> register specifies the default region index. The <code>MPU_STARTR</code> and <code>MPU_RENDER</code> register specify the regions addresses.</li> </ul> <p>This bit retains the value of a probe operation until another probe operation is issued.</p>

### 13.6.1.2 MPU Region Start Address Register, MPU\_RSTART

Address: 0x449

Access: RW in the secure mode only

**Figure 13-3 MPU\_RSTART**



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

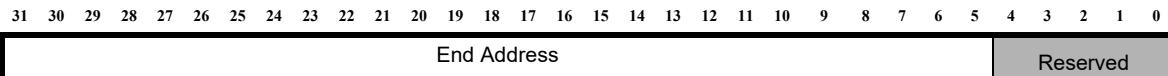
Specifies the start address of the MPU region specified in the MPU\_INDEX register. An MPU region is inclusive of the start address. The start address must be a multiple of 32 bytes. Writes to this register set the MPU\_RPER[V] bit to zero to prevent creating overlapping MPU regions.

### 13.6.1.3 MPU Region End Address Register, MPU\_RENDER

Address: 0x44A

Access: RW in the secure mode only

**Figure 13-4 MPU\_RENDER**



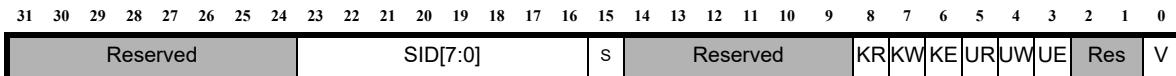
Specifies the end address of the MPU region specified in the MPU\_INDEX register. An MPU region is inclusive of the end address. The end address must be a multiple of 32 bytes. Writes to this register set the MPU\_RPER [V] bit to zero to prevent creating overlapping MPU regions.

### 13.6.1.4 MPU Region Permission Register, MPU\_RPER

Address: 0x44B

Access: RW in the secure mode only

**Figure 13-5 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.

**Table 13-4 MPU\_RPER Field Description**

Field	Bit	Description
V	[0]	<p>Region valid descriptor</p> <ul style="list-style-type: none"> <li>■ 0: region descriptor is unused.</li> <li>■ 1: region descriptor is valid.</li> </ul>
UE	[3]	<p>General Execute Permission</p> <ul style="list-style-type: none"> <li>■ 0: Instruction fetch is not permitted in the default protection region.</li> <li>■ 1: Instruction fetch is permitted in the default protection region when in user or kernel mode</li> </ul> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this UE bit relates to the secure user or kernel mode. You cannot then access the default region in the normal operating mode. If MPU_EN[15]==0, then this UE bit relates to access permissions in the both the secure and normal modes.</p>
UW	[4]	<p>General Write Permission</p> <ul style="list-style-type: none"> <li>■ 0: Default protection region is write-protected</li> <li>■ 1: Default protection region is writable when in user or kernel mode</li> </ul> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this UW bit relates to the secure user or kernel mode. You cannot then access the default region in the normal operating mode. If MPU_EN[15]==0, then this UW bit relates to access permissions in the both the secure and normal modes.</p>

**Table 13-4 MPU\_RPER Field Description**

Field	Bit	Description
UR	[5]	<p>General Read Permission</p> <ul style="list-style-type: none"> <li>■ 0: Default protection region is read-protected.</li> <li>■ 1: Default protection region is readable when in user or kernel mode</li> </ul> <p>Note that the UR field controls only read access; instruction-fetch access is controlled by the UE field.</p> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this UR bit relates to the secure user or kernel mode. You cannot then access the default region in the normal operating mode. If MPU_EN[15]==0, then this UR bit relates to access permissions in the both the secure and normal modes.</p>
KE	[6]	<p>Kernel-only Execute Permission</p> <ul style="list-style-type: none"> <li>■ 0: Instruction fetch is not permitted in default protection region</li> <li>■ 1: Instruction fetch is permitted to the default protection region when in kernel mode only</li> </ul> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this KE bit relates to the secure kernel mode only. You cannot then access the default region in the normal operating mode or the secure user mode. If MPU_EN[15]==0, then this KE bit relates to access permissions in the both the secure and normal kernel modes only.</p>
KW	[7]	<p>Kernel-only Write Permission</p> <ul style="list-style-type: none"> <li>■ 0: Default protection region is write-protected</li> <li>■ 1: Default protection region is writable when in kernel mode only</li> </ul> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this KW bit relates to the secure kernel mode only. You cannot then access the default region in the normal operating mode or the secure user mode. If MPU_EN[15]==0, then this KW bit relates to access permissions in the both the secure and normal kernel modes only.</p>
KR	[8]	<p>Kernel-only Read Permission</p> <ul style="list-style-type: none"> <li>■ 0: Default protection region is read-protected.</li> <li>■ 1: Default protection region is readable when in kernel mode only</li> </ul> <p>Note that the KR field controls only read access; instruction-fetch access is controlled by the KE field.</p> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this KR bit relates to the secure kernel mode only. You cannot then access the default region in the normal operating mode or the secure user mode. If MPU_EN[15]==0, then this KR bit relates to access permissions in the both the secure and normal kernel modes only.</p>

**Table 13-4 MPU\_RPER Field Description**

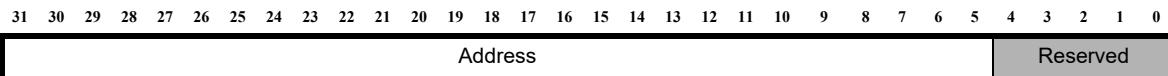
Field	Bit	Description
S	[15]	Indicates whether the default region is a secure region or normal region. <ul style="list-style-type: none"> <li>■ 0: normal region.</li> <li>■ 1: secure region.</li> </ul>
SID[7:0]	[23:16]	Default region SID field. On reset, this field is reset to 0x00. This field is valid 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.

### 13.6.2 MPU Probe Register, MPU\_PROBE

Address: 0x44C

Access: W in the secure mode only

**Figure 13-6 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 [869](#).

**Table 13-5 MPU\_PROBE Field Description**

Field	Description
Address	32-byte aligned address to probe

### 13.6.3 MPU Build Configuration Register, MPU\_BUILD

For information about the MPU\_BUILD register, see [Memory Protection Build Configuration Register, MPU\\_BUILD](#).

### 13.6.3.1 MPU Exception Cause Register, MPU\_ECR

For information about the MPU\_ECR register, see [MPU Exception Cause Register, MPU\\_ECR](#).



# 14

## Secure Debug Access

### 14.1 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 instruction 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 instruction 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.

### 14.2 Unlocking a Debug Port

When a debug port is locked, the debugger cannot access the core resources. To unlock the debug port, you must implement your own logic to unlock the debug port. When you have configured a 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*.

## 14.3 Secure Debug Unlock Module Auxiliary Register Interface

**Table 14-1 Secure Debug Registers**

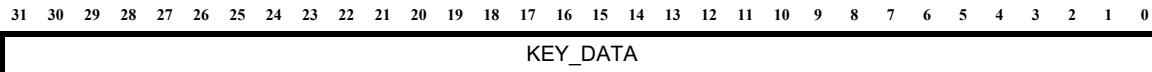
Address	Name	Access
0xF000	"Key Data Register, AUX_KEY_DATA"	Read and write by the debug host only
0xF001	"Key Unlock Register, AUX_KEY_UNLOCK"	Read and write by the debug host only

### 14.3.1 Key Data Register, AUX\_KEY\_DATA

Address: 0xF000

Access: RG

**Figure 14-1 AUX\_KEY\_DATA Register**



Use this register, as part of the debug unlock access, to read (challenge) or write (response) the key data.

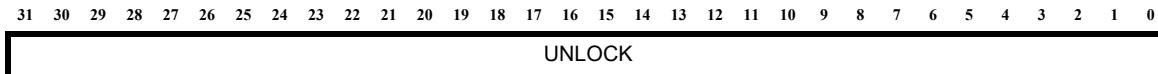
This register is included in the processor only when `scdbg_aux_unlk==true`.

### 14.3.2 Key Unlock Register, AUX\_KEY\_UNLOCK

Address: 0xF001

Access RG

**Figure 14-2 AUX\_KEY\_UNLOCK Register**



Any reads to this register triggers the unlock sequence. Any write to this register indicates that the response key has been completely written to the device and the unlock logic compares 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`.

# 15

## 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

**Table 15-1 Error Protection Auxiliary Registers**

Address	Auxiliary Register Name	Description
0x3F	<a href="#">ERP_CTRL</a>	Error protection control register
0x40	<a href="#">RFERP_STATUS_0</a>	Register File Error Protection Status Register
0x41	<a href="#">RFERP_STATUS_1</a>	Register File Error Protection Status Register
0x405	<a href="#">ESYN</a>	ECC error syndrome register
0xC7	<a href="#">ERP_BUILD</a>	Build Configuration Register: error protection

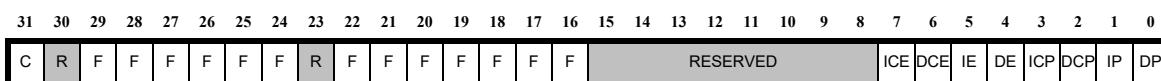
## 15.1 Error Protection Hardware Control Register, ERP\_CTRL

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

Reset 0x00000000

**Figure 15-1 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: instruction cache, data cache, ICCM0, and DCCM. Otherwise, this register is unimplemented.

When protection is disabled, 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 to 1 until they are explicitly cleared by writing to the respective bits in the register.



**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 15-2 ERP\_CTRL Field Descriptions**

Field	Bit	Description
DP	[0]	Control bit for ECC/parity protection for DCCM; When set to 1, protection is enabled. When cleared to 0, no data correction is made for single-bit errors and no exceptions are raised for double-bit, address bus, or parity errors. The fault status bits provide status in regards to errors when protection is disabled.
IP	[1]	Control bit for ECC/parity protection for ICCM0. When set to 1, protection is enabled. When cleared to 0, no data correction is made for single-bit errors and no exceptions are raised for double-bit, address bus, or parity errors. The fault status bits provide status in regards to errors when protection is disabled.
DCP	[2]	Enable ECC/parity Protection for data cache; When set to 1, protection is enabled. When cleared to 0, no data correction is made for single-bit errors and no exceptions are raised for double-bit, address bus, or parity errors. The fault status bits provide status in regards to errors when protection is disabled.

**Table 15-2 ERP\_CTRL Field Descriptions (Continued)**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
ICP	[3]	Enable ECC/parity Protection for instruction cache; When set to 1, protection is enabled. When cleared to 0, no data correction is made for single-bit errors and no exceptions are raised for double-bit, address bus, or parity errors. The fault status bits provide status in regards to errors when protection is disabled.
DE	[4]	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]	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]	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]	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.
F	[29:16] See <a href="#">Table 15-3</a>	Data and address bus faults on CCM and cache accesses.
C	[31]	Clear ECC/parity interface signals that are sticky in functionality. Set bit[31] to 1 to reset the interface signals to 0.



**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.

The fault bits in the register in [Table 15-3](#):

**Table 15-3 ERP\_CTRL Fault Bit Descriptions**

<b>Bit Position</b>	<b>Description</b>
16	Single-bit/double-bit/parity error on DCCM data word
17	Single-bit/double-bit/parity error on ICCM0 data word
18	Single-bit/double-bit/parity error on ICCM0 instruction opcode

**Table 15-3 ERP\_CTRL Fault Bit Descriptions**

Bit Position	Description
19	Single-bit/double-bit/parity error on instruction cache data instruction opcode
20	Single-bit/double-bit/parity error on data cache data word
21	Single-bit/double-bit/parity error on instruction cache tag memory
22	Single-bit/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	ITLB ECC error pending bit
29	DTLB ECC error pending bit

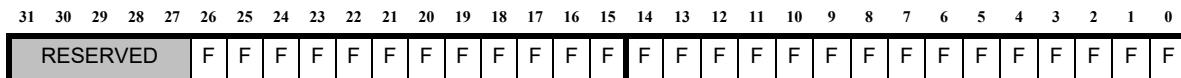
### 15.1.1 Register File Error Protection Status Register, RFERP\_STATUS\_0

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 15-2 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 15-4 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 15-4 RFERP\_STATUS\_0 Field Description**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
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.

### **15.1.2 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 15-3 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 15-5 RFERP\_STATUS\_1 Field Description**

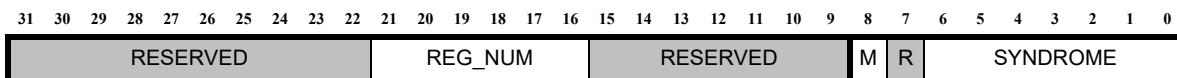
Field	Bit	Description
F	[0]	A value of 1 indicates a single-bit/double-bit/parity error on ACCL.
F	[1]	A value of 1 indicates a single-bit/double-bit/parity error on ACCH.
F	[2]	A value of 1 indicates a single-bit/double-bit/parity error on GP.
F	[3]	A value of 1 indicates a single-bit/double-bit/parity 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.

### 15.1.3 ECC Syndrome Register, EYSN

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 15-4 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.

**Table 15-6 ESYN Field Description**

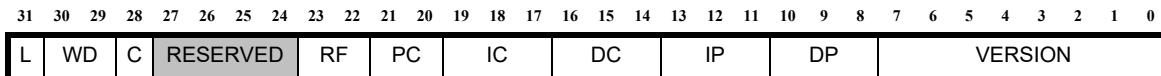
Field	Bit	Description
SYNDROME	[6:0]	Indicates the ECC syndrome.
M	[8]	If this bit is set to 1: indicates that an ECC error has occurred during a copy-back operation. This bit is present only for the processors with data cache. If data cache is not present in a processor, this bit is read as zero and ignored on writes.
REG_NUM	[21:16]	Indicates the register number that has triggered the ECC exception. If ECC protection is not configured for the processor register file, this bit is read as zero and ignored on writes.

## 15.2 Error Protection Configuration Register, ERP\_BUILD

Address: 0xC7

Access: R

**Figure 15-5 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 [23:14] are reserved in the ARC EM4 processor series.

[Table 15-7](#) lists the field descriptions.

**Table 15-7 ERP\_BUILD Register**

Field	Bit	Description
VERSION	[7:0]	Indicates the error protection version. The current value is 0x3.
DP	[10:8]	Indicates the presence of DCCM protection and its configuration. <ul style="list-style-type: none"> <li>■ 000: None</li> <li>■ 001: ECC protection on data</li> <li>■ 010: ECC protection on data and DCCM address</li> <li>■ 011: Parity protection on data</li> <li>■ 100: Parity protection on data and address</li> </ul>
IP	[13:11]	Indicates the presence of ICCM0 protection and its configuration. <ul style="list-style-type: none"> <li>■ 000: None</li> <li>■ 001: ECC protection on data</li> <li>■ 010: ECC protection on data and address</li> <li>■ 011: Parity protection on data</li> <li>■ 100: Parity protection on data and address</li> </ul>

**Table 15-7 ERP\_BUILD Register**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
DC	[16:14]	<p>Indicates presence of data cache protection and its configuration.</p> <ul style="list-style-type: none"> <li>■ 000: None</li> <li>■ 001: ECC protection on data and tag</li> <li>■ 010: ECC protection on data, tag, and index</li> <li>■ 011: Parity protection on data and tag</li> <li>■ 100: Parity protection on data, tag, and index</li> </ul>
IC	[19:17]	<p>Indicates presence of instruction cache protection and its configuration.</p> <ul style="list-style-type: none"> <li>■ 000: None</li> <li>■ 001: ECC protection on data and tag</li> <li>■ 010: ECC protection on data, tag, and index</li> <li>■ 011: Parity protection on data and tag</li> <li>■ 100: Parity protection on data, tag, and index</li> </ul>
PC	[21:20]	<p>Indicates presence of program counter protection and its configuration.</p> <ul style="list-style-type: none"> <li>■ 00: None</li> <li>■ 01: ECC protection on the program counter</li> </ul>
RF	[23:22]	<p>Indicates presence of the register file protection and its configuration.</p> <ul style="list-style-type: none"> <li>■ 00: None</li> <li>■ 01: ECC protection the register file</li> </ul>
C	[28]	<p>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 writeback to the CCM.</p> <ul style="list-style-type: none"> <li>■ 0: errors are corrected and written to the CCMs</li> <li>■ 1: errors are corrected but are not written to the CCMs</li> </ul>
WD	[30:29]	<p>Indicates the presence of the watchdog timer and its configuration.</p> <ul style="list-style-type: none"> <li>■ 00: None</li> <li>■ 01: 16-bit timer</li> <li>■ 10: 32-bit timer</li> </ul>
L	[31]	<p>Indicates presence of the lockstep interface.</p> <ul style="list-style-type: none"> <li>■ 0: lockstep interface is not available</li> <li>■ 1: lockstep interface is available</li> </ul>

### 15.2.1 Exceptions

See [Table 4-7](#) on page [226](#).



# Part 3

## Memory and System Components

### In this part:

- “Memory Protection Unit”
- “Protection Schemes”
- “ICCM”
- “Instruction Cache”
- “Instruction Fetch Queue”
- “DCCM”
- “Data Cache”
- “Program Storage”
- “Calibration Parameter Storage”
- “Cryptographic Key Storage”
- “Processor Timers”
- “External Host Debugging”
- “Debug Features”
- “Performance Counters”
- “Peripheral Bus”
- “Power Domain Management and DVFS”

- “Floating-Point Unit”
- “XY”
- “Address Generation Unit Auxiliary Interface”
- “micro DMA Controller”
- “User Auxiliary Register Interface”
- “ARC RTT”
- “Debug Features”
- “ARConnect”
- Safety Island

# 16

## Memory Protection Unit

ARCv2 ISA-based processors provide several memory-protection options, which are described in this chapter. See the Data Book for your specific processor to determine which options are available.



**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.

### 16.1 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 processors 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 user and kernel mode processes on each memory region. However, the region descriptors are backward-compatible with ARC 600. Permission is granted if an access is made in kernel mode and kernel permission is enabled, or if user permission is enabled. The user permission bits are located in the same bit-positions in the descriptor registers as they are for the ARC 600 MPU; this descriptor design allows ARC 600-style MPU descriptor to confer permissions on both user and kernel processes.

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 and the attributes that apply to a particular address in the case of overlapping regions are determined using a fixed priority scheme.

### 16.1.1 Key Features

The following are the main features of 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 can be programmed individually and independently.
- Overlapping regions are supported by a priority scheme.
- Ability to set default permissions that apply to accesses outside all programmed protection regions.
- Uses EV\_ProtV (see [Exception Vectors and the Exception Cause Register](#) on page 221) exception to indicate access violations.
- Protection exceptions are precise and can be restarted.
- Can 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.

### 16.1.2 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.

### 16.1.3 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](#).

### 16.1.4 Data Organization and Addressing

The ARCv2 address space is divided into 16 equal-sized regions that are used to define the locations of Closely-coupled Memories (ICCM and DCCM), as well as the location of peripherals and uncachable 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 writeable 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:

#### 16.1.4.1 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-writeable region in the next quarter of memory.
- A user-readable and user-writeable 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.
- A user-readable and user-writeable region for the volatile memory, including peripherals, in the last quarter. This region coincides with the volatile region specified by the AUX\_VOLATILE register (having base and limit fields) with the value 0xC0000000.

Example 1 Memory Map shows the memory map for this set up:

**Table 16-1 Example 1 -- Contiguous Regions Memory Map**

Region 0	0x00000000 0x7FFFFFFF 0x80000000	Base address: 0x00000000 Region size: 2G Regions permissions: user-execute Base address: 0x80000000
Region 1	0xBFFFFFFF 0xC0000000	Region size: 1G Regions permissions: user-read, user-write Base address: 0xC0000000
Region 2	0xFFFFFFFF	Region size: 1G 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 setup as shown in the following figures:

**Figure 16-1 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	1	

**Figure 16-2 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	0	1	0	1	

**Figure 16-3 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	1	

**Figure 16-4 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	0	1	1	0	0	1	

**Figure 16-5 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	1	

**Figure 16-6 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	0	1	1	0	0	1	

#### 16.1.4.2 Example 2 - Overlapping Regions

In this example the memory has the following features:

- The entire memory space is defined as user-readable and user-writeable 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 16-7 Overlapping Regions Memory Map**

Region 0 overlaps with Region 7	0x00000000 0x3FFFFFFF	Base address: 0x00000000 Region size: 1G Region permissions: user-execute	
Region 7	0x40000000 0xFFFFFFFF	Base address: 0x00000000 Region size: 4G Region 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 16-8 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 16-9 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	0	1	0	0	1	

**Figure 16-10 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 16-11 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	0	1	1	0	0	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 16-12 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	1	

**Figure 16-13 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	1	1	1	0	0	0	0	0	0	1	0	1	

**Figure 16-14 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 16-15 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	1	1	1	0	0	0	1	1	0	0	0	1	

**Figure 16-16 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 16-17 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	1	1	1	0	0	0	1	1	0	0	0	1	

#### 16.1.4.3 Example 2 - Overlapping Regions

In this example the memory has the following features:

- The entire memory space is defined as user-readable and user-writeable 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 16-18 Overlapping Regions Memory Map**

Region 0 overlaps with Region 7	0x00000000 0xFFFFFFFF	Base address: 0x00000000 Region size: 1G Region permissions: user-execute	
Region 7	0x40000000 0xFFFFFFFF	Base address: 0x00000000 Region size: 4G Region 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 16-19 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	1	

**Figure 16-20 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 16-21 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	1	

**Figure 16-22 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	0	1	1	0	0	1	

## 16.1.5 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](#)

### 16.1.5.1 Region Accesses

All accesses are checked against the region descriptors, according to a fixed priority, to determine what attributes to apply.

**Table 16-2 Priority Configurations**

Region	Priority (8-region configuration)	Priority (16-region configuration)
0	Highest	Highest
1	2 <sup>nd</sup> Highest	2 <sup>nd</sup> Highest
2	3 <sup>rd</sup> Highest	3 <sup>rd</sup> Highest
3	4 <sup>th</sup> Highest	4 <sup>th</sup> Highest
4	5 <sup>th</sup> Highest	5 <sup>th</sup> Highest
5	6 <sup>th</sup> Highest	6 <sup>th</sup> Highest
6	7 <sup>th</sup> Highest	7 <sup>th</sup> Highest
7	Lowest	8 <sup>th</sup> Highest
8	Not applicable	9 <sup>th</sup> Highest
9	Not applicable	10 <sup>th</sup> Highest
10	Not applicable	11 <sup>th</sup> Highest
11	Not applicable	12 <sup>th</sup> Highest
12	Not applicable	13 <sup>th</sup> Highest
13	Not applicable	14 <sup>th</sup> Highest
14	Not applicable	15 <sup>th</sup> 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.

### 16.1.5.2 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 16-3](#) lists the settings of the ECR register based on the entry to the EV\_ProtV exception.

**Table 16-3 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. This register may point to an address part-way through an instruction if that instruction and/or its long-immediate data straddles two 32-bit words.

In the case of data access violations, EFA is set to the address of the data access that caused the violation.

### 16.1.5.3 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. Hence, 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 table below:

**Table 16-4 ECR parameter values for EV\_ProtV sources**

Source of EV_ProtV	ECR Parameter Value
Code protection	0x01
Stack Checking	0x02
MPU	0x04

Hence, 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, and thus the ECR parameter value is 0x4 (MPU only).

#### 16.1.5.4 Caution When Updating Region Settings

As the MPU protects all the memory space on 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.

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 into 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.

#### 16.1.6 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 16-5](#) lists the MPU auxiliary registers.

**Table 16-5 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
0x42a	MPU_RDB4	MPU region descriptor base 4
0x42b	MPU_RDP4	MPU region descriptor permissions 4

**Table 16-5 MPU Register Set**

<b>Address</b>	<b>Auxiliary Register Name</b>	<b>Description</b>
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

### 16.1.7 MPU Enable Register, MPU\_EN

Address: 0x409

Access: RW

- 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.

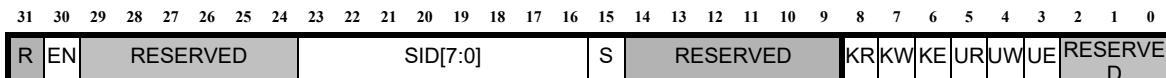
#### When secure MPU is not configured:

This register contains 0x00000000 on reset.

#### When secure MPU is configured:

For a build without the `-mpu_sid_option`, the reset value is 0x4000\_81C00 and for a build with `-mpu_sid_option==true`, the reset value is 0x40FF\_81C0.

**Figure 16-23 MPU\_EN Register**



**Table 16-6 MPU Enable Register**

Field	Bit	Description
UE	[3]	<p>General Execute Permission</p> <ul style="list-style-type: none"> <li>■ 0: Instruction fetch is not permitted in the default protection region.</li> <li>■ 1: Instruction fetch is permitted in the default protection region when in user or kernel mode</li> </ul> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this UE bit relates to the secure user or kernel mode. You cannot then access the default region in the normal operating mode.</p> <p>If MPU_EN[15]==0, then this UE bit relates to access permissions in the both the secure and normal modes.</p>

**Table 16-6 MPU Enable Register**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
UW	[4]	<p>General Write Permission</p> <ul style="list-style-type: none"> <li>■ 0: Default protection region is write-protected</li> <li>■ 1: Default protection region is writable when in user or kernel mode</li> </ul> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this UW bit relates to the secure user or kernel mode. You cannot then access the default region in the normal operating mode.</p> <p>If MPU_EN[15]==0, then this UW bit relates to access permissions in the both the secure and normal modes.</p>
UR	[5]	<p>General Read Permission</p> <ul style="list-style-type: none"> <li>■ 0: Default protection region is read-protected.</li> <li>■ 1: Default protection region is readable when in user or kernel mode</li> </ul> <p>Note that the UR field controls only read access; instruction-fetch access is controlled by the UE field.</p> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this UR bit relates to the secure user or kernel mode. You cannot then access the default region in the normal operating mode.</p> <p>If MPU_EN[15]==0, then this UR bit relates to access permissions in the both the secure and normal modes.</p>
KE	[6]	<p>Kernel-only Execute Permission</p> <ul style="list-style-type: none"> <li>■ 0: Instruction fetch is not permitted in default protection region</li> <li>■ 1: Instruction fetch is permitted to the default protection region when in kernel mode only</li> </ul> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this KE bit relates to the secure kernel mode only. You cannot then access the default region in the normal operating mode or the secure user mode.</p> <p>If MPU_EN[15]==0, then this KE bit relates to access permissions in the both the secure and normal kernel modes only.</p>
KW	[7]	<p>Kernel-only Write Permission</p> <ul style="list-style-type: none"> <li>■ 0: Default protection region is write-protected</li> <li>■ 1: Default protection region is writable when in kernel mode only</li> </ul> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this KW bit relates to the secure kernel mode only. You cannot then access the default region in the normal operating mode or the secure user mode.</p> <p>If MPU_EN[15]==0, then this KW bit relates to access permissions in the both the secure and normal kernel modes only.</p>

**Table 16-6 MPU Enable Register**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
KR	[8]	<p>Kernel-only Read Permission</p> <ul style="list-style-type: none"> <li>■ 0: Default protection region is read-protected.</li> <li>■ 1: Default protection region is readable when in kernel mode only</li> </ul> <p>Note that the KR field controls only read access; instruction-fetch access is controlled by the KE field.</p> <p><b>Note:</b> the secure or normal mode is defined by the MPU_EN[15] bit. If MPU_EN[15]==1, then this KR bit relates to the secure kernel mode only. You cannot then access the default region in the normal operating mode or the secure user mode.</p> <p>If MPU_EN[15]==0, then this KR bit relates to access permissions in the both the secure and normal kernel modes only.</p>
S	[15]	<p>Indicates whether the default region is a secure region or normal region.</p> <ul style="list-style-type: none"> <li>■ 0: normal region.</li> <li>■ 1: secure region.</li> </ul> <p>This field is only valid with MPU version 0x04.</p>
SID[7:0]	[23:16]	<p>Default region SID field; the default value is 0x1.</p> <p>This field is only valid with MPU version 0x04. If <code>-mpu_sid_option==false</code>, these bits are read as zero and ignored on writes.</p>
EN	[30]	MPU enable bit - always enabled

### 16.1.8 MPU Exception Cause Register, MPU\_ECR

Address: 0x420

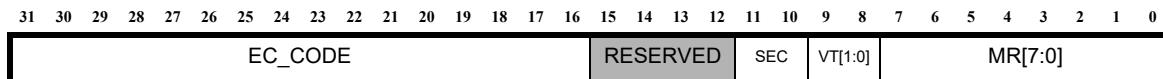
Access: R

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 Exception Cause Register (ECR) 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, etc). 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 16-24 MPU\_ECR Register**



**Table 16-7 MPU Exception Cause Register**

Field	Bit	Description
MR	[7:0]	<p>MPU Region</p> <p>Indicates which protection region was violated to cause the interrupt according to the following values:</p> <ul style="list-style-type: none"> <li>0x00 MPU region 0</li> <li>0x01 MPU region 1</li> <li>0x02 MPU region 2</li> <li>0x03 MPU region 3</li> <li>0x04 MPU region 4</li> <li>0x05 MPU region 5</li> <li>0x06 MPU region 6</li> <li>0x07 MPU region 7</li> <li>0x08 MPU region 8</li> <li>0x09 MPU region 9</li> <li>0x0a MPU region 10</li> <li>0x0b MPU region 11</li> <li>0x0c MPU region 12</li> <li>0x0d MPU region 13</li> <li>0x0e MPU region 14</li> <li>0x0f MPU region 15</li> <li>0xff Access missed all regions</li> </ul>
VT	[9:8]	<p>Violation Type</p> <p>Indicates the type of access that caused the violation according to the following values:</p> <ul style="list-style-type: none"> <li>■ 00: Execution Violation</li> <li>■ 01: Data Read Violation</li> <li>■ 10: Data Write Violation</li> <li>■ 11: Data Read-modify-write Violation (such as EX instruction)</li> </ul>
SEC	[11:10]	<p>Indicates if a secure violation has occurred. This field is only valid with MPU version 0x04.</p> <ul style="list-style-type: none"> <li>■ 00: No secure violation has occurred.</li> <li>■ 01: Secure violation has occurred; secure resources are being accessed in the normal mode.</li> <li>■ 10: An SID mismatch has occurred.</li> <li>■ 11: An SID and secure mode mismatch have occurred.</li> </ul>
EC_CODE[15:0]	[31:16]	<p>Exception Cause Code</p> <p>Set to 0x0006 for compatibility with the ARCV2 Ev_ProtV exception cause.</p>

### 16.1.9 MPU Region Descriptor Base Registers, MPU\_RDB0 to MPU\_RDB15

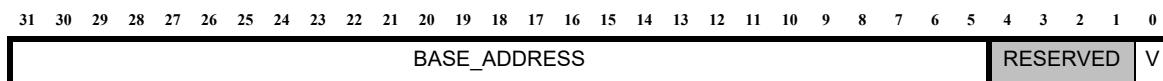
The MPU region descriptor base registers, MPU\_RDB0 to MPU\_RDB15, contain the base address for each region and enable the particular region. These registers contain 0x00000000 on reset.

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 16-25 MPU Region Descriptor Base Registers**



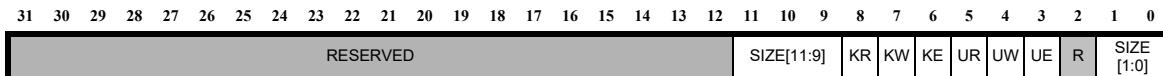
**Table 16-8 MPU Region Descriptor Base Registers Field Description**

Field	Bit	Description
V	[0]	Region Descriptor Validity Flag <ul style="list-style-type: none"> <li>■ 0: This region descriptor is unused.</li> <li>■ 1: This region descriptor is valid.</li> </ul>
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 supports a minimum protection region size of 32 bytes.

### 16.1.10 MPU Region Descriptor Permissions Registers, MPU\_RDP0 to MPU\_RDP15

The MPU region descriptor permissions registers, MPU\_RDP0 to MPU\_RDP15, set the size and set the permissions for each region. This register contains 0x00000000 on reset.

**Figure 16-26 MPU Region Descriptor Permissions Registers**



**Table 16-9 MPU Region Descriptor Permission Registers Field Description**

Field	Bit	Description																																																										
SIZE	[11:9] and [1:0]	<p>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:</p> <p>Note: For MPU version 0x03, minimum region size is 32 bytes. For MPU version 0x02, minimum region size is 2 Kbytes.</p> <table> <tr><td>00000-00011</td><td>Reserved</td></tr> <tr><td>00100</td><td>32</td><td>00101</td><td>64</td></tr> <tr><td>00110</td><td>128</td><td>00111</td><td>256</td></tr> <tr><td>01000</td><td>512</td><td>01001</td><td>1k</td></tr> <tr><td>01010</td><td>2K</td><td>01011</td><td>4K</td></tr> <tr><td>01100</td><td>8K</td><td>01101</td><td>16K</td></tr> <tr><td>01110</td><td>32K</td><td>01111</td><td>64K</td></tr> <tr><td>10000</td><td>128K</td><td>10001</td><td>256K</td></tr> <tr><td>10010</td><td>512K</td><td>10011</td><td>1M</td></tr> <tr><td>10100</td><td>2M</td><td>10101</td><td>4M</td></tr> <tr><td>10110</td><td>8M</td><td>10111</td><td>16M</td></tr> <tr><td>11000</td><td>32M</td><td>11001</td><td>64M</td></tr> <tr><td>11010</td><td>128M</td><td>11011</td><td>256M</td></tr> <tr><td>11100</td><td>512M</td><td>11101</td><td>1G</td></tr> <tr><td>11110</td><td>2G</td><td>11111</td><td>4G</td></tr> </table>	00000-00011	Reserved	00100	32	00101	64	00110	128	00111	256	01000	512	01001	1k	01010	2K	01011	4K	01100	8K	01101	16K	01110	32K	01111	64K	10000	128K	10001	256K	10010	512K	10011	1M	10100	2M	10101	4M	10110	8M	10111	16M	11000	32M	11001	64M	11010	128M	11011	256M	11100	512M	11101	1G	11110	2G	11111	4G
00000-00011	Reserved																																																											
00100	32	00101	64																																																									
00110	128	00111	256																																																									
01000	512	01001	1k																																																									
01010	2K	01011	4K																																																									
01100	8K	01101	16K																																																									
01110	32K	01111	64K																																																									
10000	128K	10001	256K																																																									
10010	512K	10011	1M																																																									
10100	2M	10101	4M																																																									
10110	8M	10111	16M																																																									
11000	32M	11001	64M																																																									
11010	128M	11011	256M																																																									
11100	512M	11101	1G																																																									
11110	2G	11111	4G																																																									
UE	[3]	<p>General Execute Permission</p> <ul style="list-style-type: none"> <li>■ 0: Instruction fetch is not permitted</li> <li>■ 1: Instruction fetch is permitted when in user or kernel mode</li> </ul>																																																										
UW	[4]	<p>General Write Permission</p> <ul style="list-style-type: none"> <li>■ 0: Region is write-protected</li> <li>■ 1: Region is writeable when in user or kernel mode</li> </ul>																																																										

**Table 16-9 MPU Region Descriptor Permission Registers Field Description**

Field	Bit	Description
UR	[5]	<p>General Read Permission</p> <ul style="list-style-type: none"> <li>■ 0: Region is read-protected.</li> <li>■ 1: Region is readable when in user or kernel mode</li> </ul> <p>Note that the UR field controls only read access; instruction-fetch access is controlled by the UE field.</p>
KE	[6]	<p>Kernel-only Execute Permission</p> <ul style="list-style-type: none"> <li>■ 0: Instruction fetch is not permitted</li> <li>■ 1: Instruction fetch is permitted granted when in kernel mode only</li> </ul>
KW	[7]	<p>Kernel-only Write Permission</p> <ul style="list-style-type: none"> <li>■ 0: Region is write-protected</li> <li>■ 1: Region is writeable when in kernel mode only</li> </ul>
KR	[8]	<p>Kernel-only Read Permission</p> <ul style="list-style-type: none"> <li>■ 0: Region is read-protected.</li> <li>■ 1: Region is readable when in kernel mode only</li> </ul> <p>Note that the KR field controls only read access; instruction-fetch access is controlled by the KE field.</p>


**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 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.

**Table 16-10 Build Configuration Registers**

Number	Name	Access	Description
0x6D	Memory Protection Build Configuration Register, MPU_BUILD	R	Build configuration for: memory protection unit

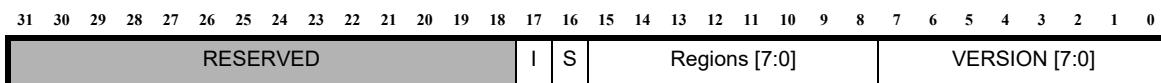
### 16.1.11 Memory Protection Build Configuration Register, MPU\_BUILD

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 16-27 MPU\_BUILD Register**



**Table 16-11 MPU\_BUILD Field Description**

Field	Bit	Description
VERSION	[7:0]	<p>Version</p> <p>Indicates the version of the MPU:</p> <ul style="list-style-type: none"> <li>0x00 : when no MPU is present</li> <li>0x02 : MPU with 2 Kbytes minimum region size</li> <li>0x03 : MPU with 32 bytes minimum region size</li> <li>0x04 : supports defining regions as secure and normal, added SID protection, indirect programming interface; when -sec_modes_option==true, the MPU regions are a minimum of 32 bytes, and thereafter are multiples of 32 bytes.</li> </ul> <p>All other values are reserved.</p>
Regions	[15:8]	<p>Number of Regions</p> <p>Indicates the number of regions supported:</p> <ul style="list-style-type: none"> <li>■ 0x00 when no MPU is present</li> <li>■ 0x01 for 1 region</li> <li>■ 0x02 for 2 regions</li> <li>■ 0x04 for 4 regions</li> <li>■ 0x08 for 8 regions</li> <li>■ 0x10 for 16 regions</li> </ul> <p>All other values are reserved.</p>
S	[16]	MPU supports defining regions as secure or normal. This bit exists only when MPU version is 0x04.

**Table 16-11 MPU\_BUILD Field Description**

Field	Bit	Description
I	[17]	MPU supports SID mechanism. This bit exists only when MPU version is 0x04. <ul style="list-style-type: none"><li>■ 0x0: SID is not supported; -mpu_sid_option==false</li><li>■ 0x1: SID is supported; -mpu_sid_option==true</li></ul>

### 16.1.12 MPU Exceptions

See [Table 4-7](#) on page [226](#).

# 17

## Protection Schemes

ARCv2 ISA-based processors provide several memory-protection options, which are described in this chapter. See the Data Book 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.

### 17.1 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. 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.

### 17.1.1 Build Configuration Register

[Stack Region Configuration Register, STACK\\_REGION\\_BUILD](#) (0xC5) identifies the presence and version of stack checking in the build.

### 17.1.2 Enabling Stack Checking

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

### 17.1.3 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.



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.

### 17.1.4 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.

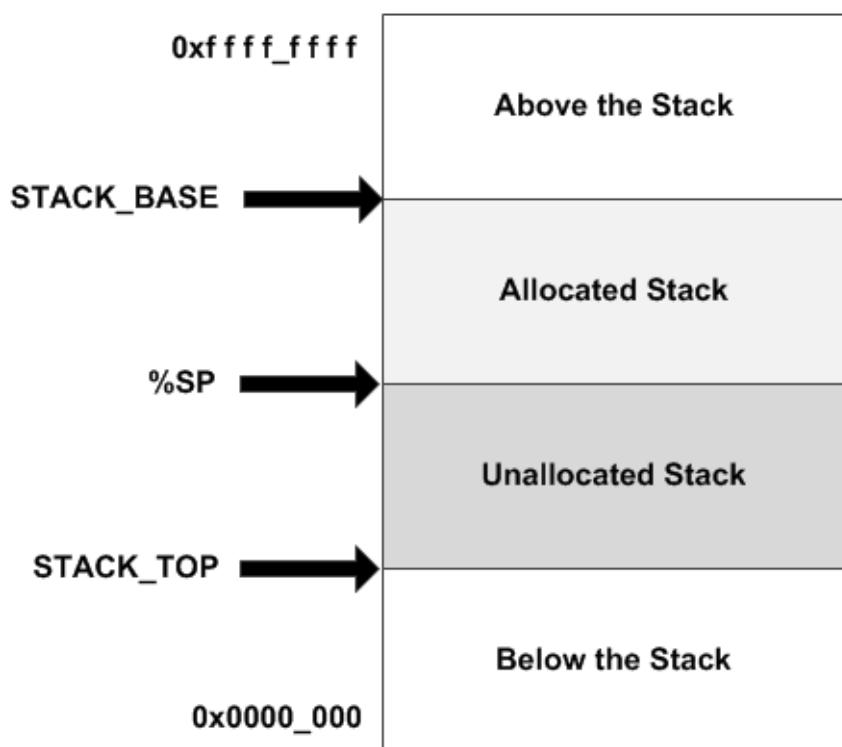
#### 17.1.4.1 Stack Regions

Figure 17-1 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 hence 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 hence 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 17-1 Stack regions**

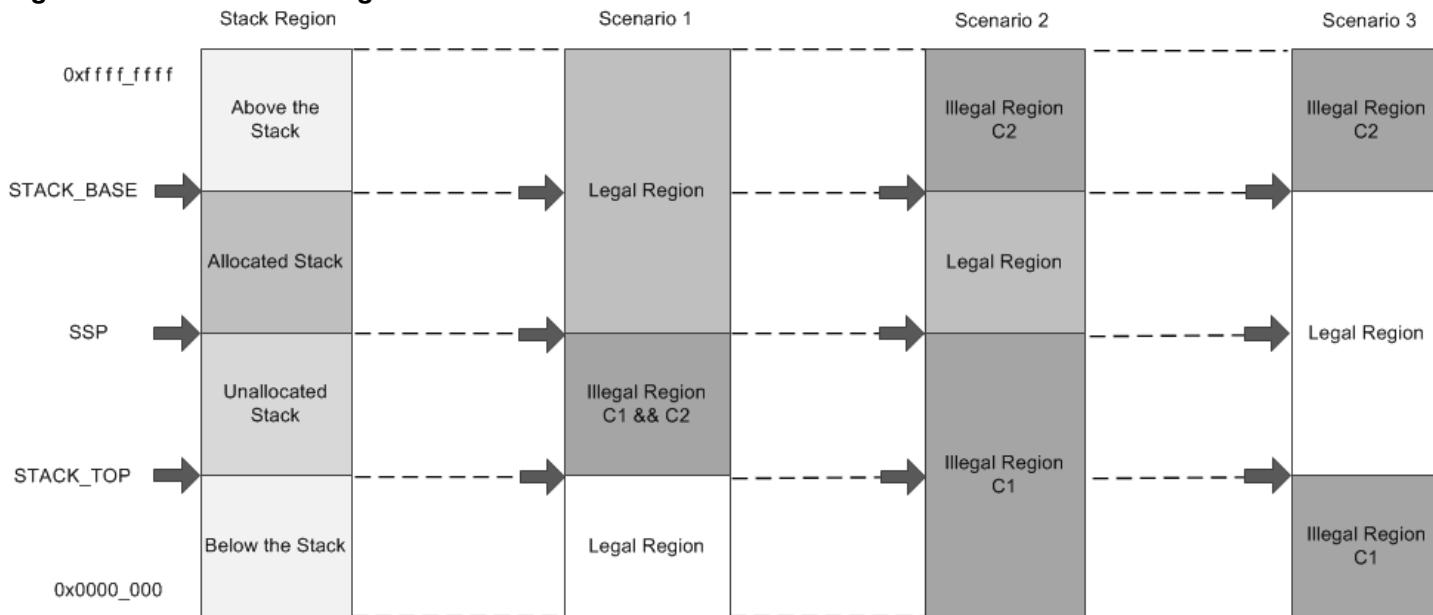
### 17.1.4.2 Stack Access Rules

- 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 17-2](#) and [Figure 17-2](#) illustrate the conditions checked in three possible scenarios.

**Figure 17-2 Stack Checking Scenarios**



**Table 17-1 Stack Checking Specification**

Conditions	Scenario 1 Non-%SP-Based Address. <b>Example:</b> LD r0, [r1];	Scenario 2 %SP-Based, No Auto-Update of %SP <b>Example:</b> LD r0, [r28];	Scenario 3 %SP-Based, with Auto-Update of %SP <b>Example:</b> 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

### 17.1.5 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 4-7](#) on page [226](#)

### 17.1.6 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.

### 17.1.7 Stack Region Auxiliary Register Set

**Table 17-2 Stack Region Checking Auxiliary Registers**

Auxiliary Register	Name	Description
0x260	USTACK_TOP (see <a href="#">STACK_TOP</a> )	User stack top address register
0x261	USTACK_BASE (see <a href="#">STACK_BASE</a> )	User stack base address register
0x262	S_USTACK_TOP (see <a href="#">S_STACK_TOP</a> )	Secure mode user stack top address register
0x263	S_USTACK_BASE (see <a href="#">S_STACK_BASE</a> )	Secure mode user stack base address register
0x264	KSTACK_TOP (see <a href="#">STACK_TOP</a> )	Kernel stack top address register
0x265	KSTACK_BASE (see <a href="#">STACK_BASE</a> )	Kernel stack base address register
0x266	S_KSTACK_TOP (see <a href="#">S_STACK_TOP</a> )	Secure mode kernel stack top address register
0x267	S_KSTACK_BASE (see <a href="#">S_STACK_BASE</a> )	Secure mode kernel stack base address register

### 17.1.7.1 Stack Region Top Address, STACK\_TOP

USTACK\_TOP Address: 0x260

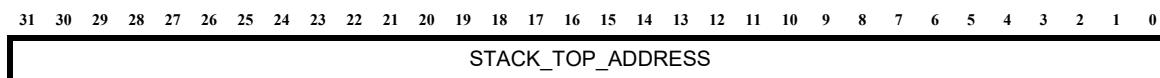
KSTACK\_TOP Address: 0x264

Access RW

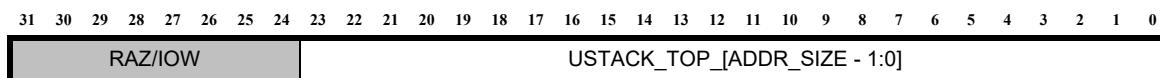
You can use the stack region top address register, STACK\_TOP to set the top address of the valid stack region in memory.

You can use the following auxiliary registers to set the stack region in the normal user and kernel modes. These auxiliary registers are present in the processor if you have configured the processor to include stack checking.

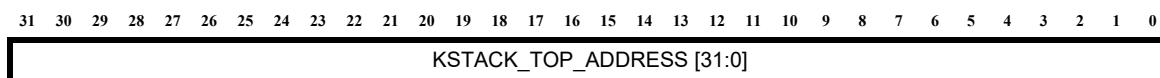
**Figure 17-3 USTACK\_TOP when ADDR\_SIZE == 32**



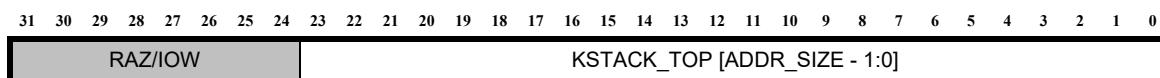
**Figure 17-4 USTACK\_TOP when ADDR\_SIZE < 32**



**Figure 17-5 KSTACK\_TOP when ADDR\_SIZE == 32**



**Figure 17-6 KSTACK\_TOP, when ADDR\_SIZE < 32**



All load and store addresses are checked against the region between STACK\_BASE and STACK\_TOP. Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store is between the SP and STACK\_TOP region, a protection violation exception, EV\_ProvV, 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\_ProvV, occurs indicating that an invalid stack pointer (SP) address was used.

### 17.1.7.2 Stack Region Base Address, STACK\_BASE

USTACK\_BASE Address: 0x261

KSTACK\_BASE Address: 0x265

Access RW

The stack region base address register, STACK\_BASE, is a read or write register, and is used to set the base address of the valid stack region in memory.

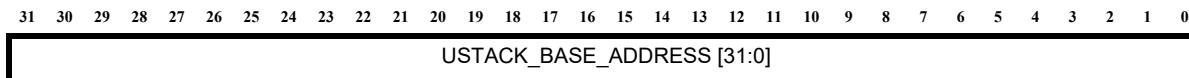
You can use the following auxiliary registers to set the stack region in user mode and kernel mode. These auxiliary registers are present in the processor if you have configured the processor to include stack checking.

You can use the following auxiliary registers to set the stack region in the normal user and kernel modes. These auxiliary registers are present in the processor if you have configured the processor to include stack checking.

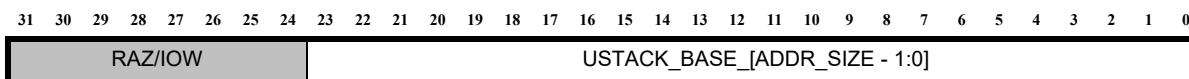


**Note** The stack grows downward from higher memory address to a lower memory address.

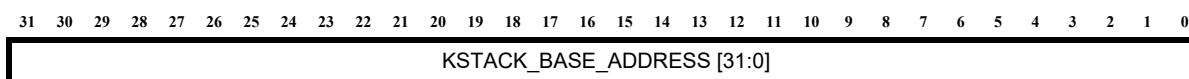
**Figure 17-7 USTACK\_BASE when ADDR\_SIZE == 32**



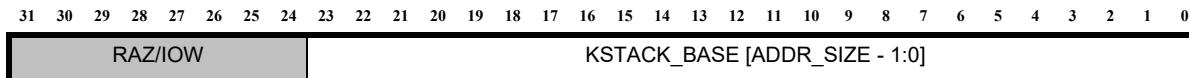
**Figure 17-8 USTACK\_BASE when ADDR\_SIZE < 32**



**Figure 17-9 KSTACK\_BASE when ADDR\_SIZE == 32**



**Figure 17-10 KSTACK\_BASE when ADDR\_SIZE < 32**



All load and store addresses are checked against the region between STACK\_BASE and STACK\_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 STACK\_TOP region, a protection violation exception, EV\_ProvV, 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\_ProfV, occurs indicating that an invalid stack pointer (SP) address was used.

### 17.1.7.3 Secure Stack Region Top Address, S\_STACK\_TOP

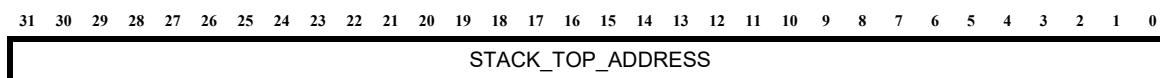
S\_USTACK\_TOP Address: 0x262

S\_KSTACK\_TOP Address: 0x266

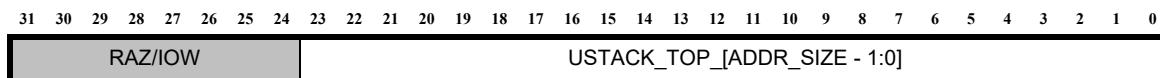
Access RW in the secure mode only

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.

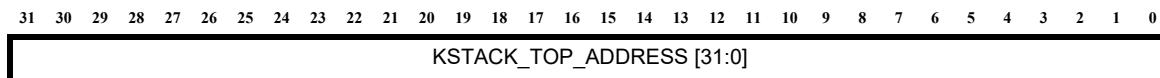
**Figure 17-11 S\_USTACK\_TOP when ADDR\_SIZE == 32**



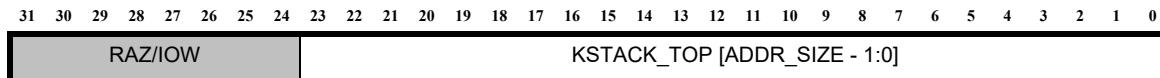
**Figure 17-12 S\_USTACK\_TOP when ADDR\_SIZE < 32**



**Figure 17-13 S\_KSTACK\_TOP when ADDR\_SIZE == 32**



**Figure 17-14 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\_ProfV, 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\_ProfV, occurs indicating that an invalid stack pointer (SP) address was used.

### 17.1.7.4 Secure Stack Region Base Address, S\_STACK\_BASE

S\_USTACK\_BASE Address: 0x263

S\_KSTACK\_BASE Address: 0x267

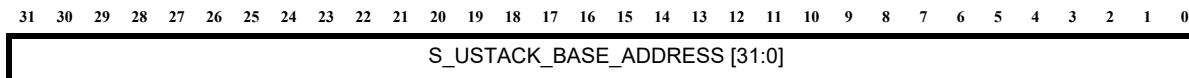
Access RW in the secure mode only

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`.

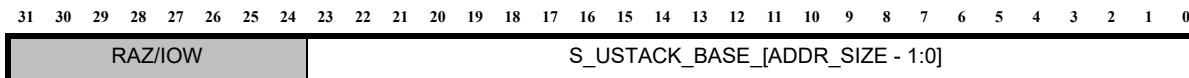


**Note** The stack grows downward from higher memory address to a lower memory address.

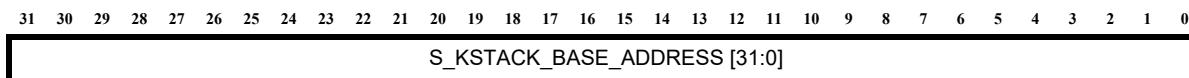
**Figure 17-15 S\_USTACK\_BASE when ADDR\_SIZE == 32**



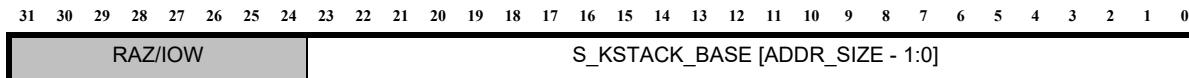
**Figure 17-16 S\_USTACK\_BASE when ADDR\_SIZE < 32**



**Figure 17-17 S\_KSTACK\_BASE when ADDR\_SIZE == 32**



**Figure 17-18 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\_ProvV, 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\_ProvV, occurs indicating that an invalid stack pointer (SP) address was used.

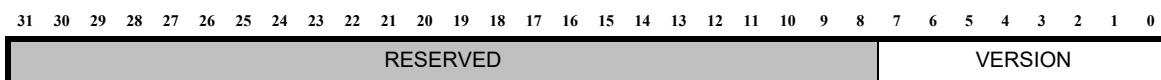
### 17.1.7.5 Stack Region Configuration Register, STACK\_REGION\_BUILD

Address: 0xC5

Access: R

The stack region configuration register, STACK\_REGION\_BUILD, indicates the presence of the stack region registers, [Stack Region Base Address, STACK\\_BASE](#) and [Stack Region Top Address, STACK\\_TOP](#).

**Figure 17-19 STACK\_REGION\_BUILD Register**



**Table 17-3 STACK\_REGION\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of stack region checking</b> <ul style="list-style-type: none"><li>■ 0x01: ARC 700 style stack checking</li><li>■ 0x02: ARCv2 style stack checking</li></ul>

### 17.1.8 Stack Checking Exceptions

See [Table 4-7](#) on page [226](#).

## 17.2 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

Hence, multiple protection violations can be reported by raising a single exception. See the [Table 4-7 Exception Vectors and Cause Codes](#) for more information about the exception information for each of the protection schemes.

## 17.3 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.

### 17.3.1 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.

### 17.3.2 Code Protection Behavior

This section describes how the code protection scheme behaves in relation to specific scenarios.

#### 17.3.2.1 Processor Reset

Protection is valid immediately after reset.

#### 17.3.2.2 User and Kernel Mode

The code protection scheme applies to both user and kernel modes.

#### 17.3.2.3 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.

#### 17.3.2.4 DMI Access

DMI access violations do not raise code-protection exceptions. Write commands from DMI are ignored, and read transactions return zero data.

#### 17.3.2.5 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.

### 17.3.2.6 DFT Scan

Code protection is not available when the core is accessed through the DFT (Design-for-Test) scan chain.

### 17.3.2.7 Code Protection Exception

The EV\_ProtV exception is taken, the same as other data-access protection violations.

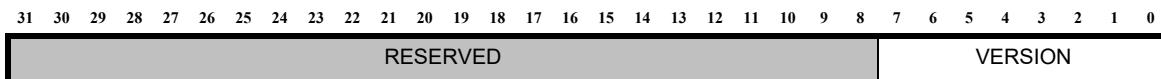
For details on the ECR register, see the [Exception Cause Register, ECR](#).

### 17.3.3 Code Protection Register, CPROT\_BUILD

Address: 0xC9

Access: R

**Figure 17-20 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 17-4 CPROT\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Code protection feature</b> <ul style="list-style-type: none"><li>■ 0x0: code protection feature is disabled</li><li>■ 0x1: code protection feature is enabled</li></ul>

# 18

## ICCM

### 18.1 ICCM Introduction

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 18-1 ICCM Registers**

Address	Auxiliary Register Name	Description
0x208	<a href="#">AUX_ICCM</a>	Base address of multiple ICCMs
0x78	<a href="#">ICCM_BUILD</a>	Build configuration register for ICCM

## 18.2 ICCM base address, AUX\_ICCM

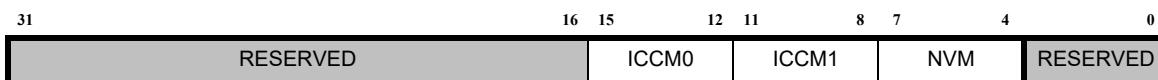
Address: 0x208

- Access:
- When SecureShield 2+2 mode is not configured: RW
  - When SecureShield 2+2 mode is configured: RW in the secure mode only

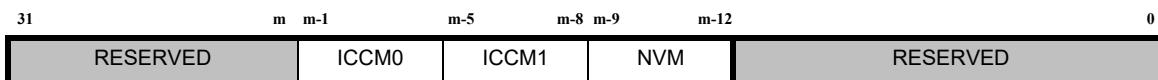
**Figure 18-1 AUX\_ICCM, base address for ICCM (ADDR\_SIZE == 32)**



**Figure 18-2 AUX\_ICCM, base address for ICCM (ADDR\_SIZE == 16)**



**Figure 18-3 AUX\_ICCM, base address for ICCM (ADDR\_SIZE == m)**



If an ICCM0, ICCM1, or NVM 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 NVM is configured, the AUX\_ICCM.NVM field identifies the region that contains the NVM. 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 Icache, IFQ, ICCM0, and ICCM1 do not exist in the processor design, NV\_ICCM is always 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.

## 18.3 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 18-4 ICCM\_BUILD Register**



**Table 18-2 ICCM\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	Current version 0x5; supports configuring ICCM0 and ICCM1 into secure and normal regions.
ICCM0_SIZE	[11:8]	Size of ICCM0 RAM <ul style="list-style-type: none"> <li>■ 0x0: Not present</li> <li>■ 0x1: 512B</li> <li>■ 0x2: 1KB</li> <li>■ 0x3: 2KB</li> <li>■ 0x4: 4KB</li> <li>■ 0x5: 8KB</li> <li>■ 0x6: 16KB</li> <li>■ 0x7: 32KB</li> <li>■ 0x8: 64KB</li> <li>■ 0x9: 128KB</li> <li>■ 0xA: 256KB</li> <li>■ 0xB: 512KB</li> <li>■ 0xC: 1MB</li> <li>■ 0xD: 2 MB</li> </ul>

**Table 18-2 ICCM\_BUILD Field Descriptions (Continued)**

Field	Bit	Description
ICCM1_SIZE	[15:12]	<p>Size of ICCM1 RAM</p> <ul style="list-style-type: none"> <li>■ 0x0: Not present</li> <li>■ 0x1: 512B</li> <li>■ 0x2: 1KB</li> <li>■ 0x3: 2KB</li> <li>■ 0x4: 4KB</li> <li>■ 0x5: 8KB</li> <li>■ 0x6: 16KB</li> <li>■ 0x7: 32KB</li> <li>■ 0x8: 64KB</li> <li>■ 0x9: 128KB</li> <li>■ 0xA: 256KB</li> <li>■ 0xB: 512KB</li> <li>■ 0xC: 1MB</li> <li>■ 0xD: 2MB</li> </ul>
NVM_SIZE	[27:24]	<ul style="list-style-type: none"> <li>■ 0x0: Not present</li> <li>■ 0x1 to 0x2: Reserved</li> <li>■ 0x3: 2KB</li> <li>■ 0x4: 4KB</li> <li>■ 0x5: 8KB</li> <li>■ 0x6: 16KB</li> <li>■ 0x7: 32KB</li> <li>■ 0x8: 64KB</li> <li>■ 0x9: 128KB</li> <li>■ 0xA: 256KB</li> </ul>
NI	[28]	<p>Indicates how NV_ICCM is implemented</p> <ul style="list-style-type: none"> <li>■ 0: IFQ is not implemented for NV_ICCM</li> <li>■ 1: IFQ is implemented for NV_ICCM</li> </ul>

## 18.4 Exceptions

See [Table 4-7](#) on page [226](#).

# 19

## Instruction Fetch Queue

### 19.1 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.

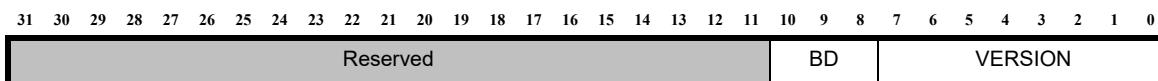
#### 19.1.1 Instruction Fetch Queue Configuration Register, IFQUEUE\_BUILD

Address: 0xFE

Access: R

The Instruction Fetch Queue Configuration register (IFQUEUE\_BUILD) contains the version of the Instruction Fetch Queue.

**Figure 19-1 IFQUEUE\_BUILD Register**



**Table 19-1 IFQUEUE\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of Instruction Fetch Queue</b> 0x02
BD[2:0]	[10:8]	<b>Instruction Fetch Queue entries</b> <ul style="list-style-type: none"> <li>■ 0x0: 1 entry</li> <li>■ 0x2: 4 entries</li> <li>■ 0x3: 8 entries</li> </ul> All other values are reserved.



# 20

## Instruction Cache

### 20.1 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 in order 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.

### 20.2 Auxiliary Registers

**Table 20-1 Basic Instruction Cache Auxiliary Registers (IC\_FEATURE\_LEVEL==0)**

Address	Auxiliary Register Name	Description
0x10	<a href="#">Invalidate Instruction Cache, IC_IVIC</a>	Invalidate instruction cache
0x11	<a href="#">Instruction Cache Control Register, IC_CTRL</a>	Instruction cache control register

**Table 20-2 Instruction Cache Auxiliary Registers (IC\_FEATURE\_LEVEL==1)**

<b>Address</b>	<b>Auxiliary Register Name</b>	<b>Description</b>
0x13	<a href="#">Lock Instruction Cache Line, IC_LIL</a>	Lock instruction cache line
0x19	<a href="#">Invalidate Instruction-Cache Line, IC_IVIL</a>	Invalidate instruction cache line

**Table 20-3 Instruction Cache Auxiliary Registers (IC\_FEATURE\_LEVEL==2)**

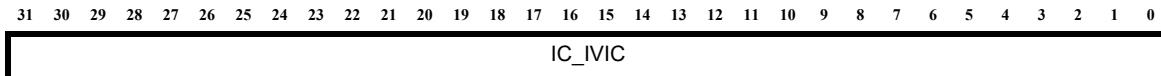
<b>Address</b>	<b>Auxiliary Register Name</b>	<b>Description</b>
0x1A	<a href="#">Instruction Cache External-Access Address, IC_RAM_ADDR</a>	Instruction cache external access RAM address
0x1B	<a href="#">Instruction-Cache Tag Access, IC_TAG</a>	Instruction cache tag access
0x1C	<a href="#">Instruction Cache Secure Bit Tag Register, IC_XTAG</a>	Indicates the secure mode of a cache line in the instruction cache tag
0x1D	<a href="#">Instruction Cache Data Access, IC_DATA</a>	Instruction cache data access

## 20.2.1 Invalidate Instruction Cache, IC\_IVIC

Address: 0x10

Access: W

**Figure 20-1 IC\_IVIC Register**



### When SecureShield 2+2 mode is not configured

A write to the IC\_IVIC register invalidates and unlocks 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 20-4 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 and unlocks 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

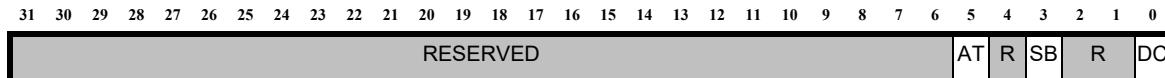
The IC\_CTRL.SB bit is not updated when the instruction cache is successfully invalidated.

## 20.2.2 Instruction Cache Control Register, IC\_CTRL

Address: 0x11

Access: RW

**Figure 20-2 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 20-5 IC\_CTRL Control Flags**

Flag Name	Bit	Description	Access Type
DC	[0]	Disable Cache: Enables/Disables the cache ■ 0 – Enable Cache ■ 1 – Disable Cache	R/W
SB	[3]	Success Bit: Success of last cache operation ■ 0 – Last cache operation failed ■ 1 – Last cache operation succeeded	R/W
AT	[5]	Address Debug Type: Used for debug purposes for when accessing cache RAMs. ■ 0 – Direct Cache RAM Access ■ 1 – Cache Controlled RAM Access	R/W

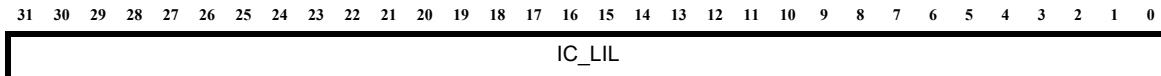
For more information about the IC\_CTRL register implementation, see your *ARCv2-based processor databook*.

### 20.2.3 Lock Instruction Cache Line, IC\_LIL

Address: 0x13

Access: W

**Figure 20-3 IC\_LIL Register**



#### When SecureShield 2+2 mode is not configured

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 20-6 IC\_LIL Description**

Operating Mode	Behavior
Core or debugger is in the normal mode	<ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> </ul>

**Table 20-6 IC\_LIL Description**

Operating Mode	Behavior
Core or debugger is in the secure mode	<ul style="list-style-type: none"> <li>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.</li> </ul>



**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.



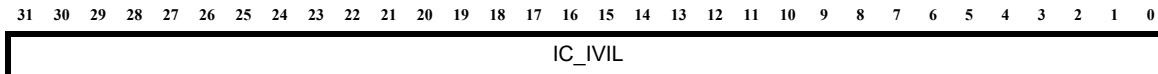
**Caution** Be aware that locking all the ways of a set prevents any further areas of memory from being cached if they map to the same set. Line-locking uncacheable regions might result in unpredictable memory accesses.

## 20.2.4 Invalidate Instruction-Cache Line, IC\_IVIL

Address: 0x19

Access: W

**Figure 20-4 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

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 20-7 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).
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).



All SR accesses to the IC\_IVIL register must be followed by three NOP instructions.

## 20.2.5 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



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 20-8 Cache Line Access in the Debug Unlocked in the Normal Mode**

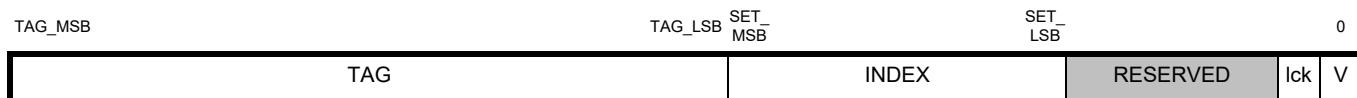
	Cache Line is in Secure Memory Space	Cache Line is in the Normal Memory Space
AT==0; and the cache line is a hit	The IC_TAG, IC_XTAG, and IC_DATA registers are not updated. The Success bit is cleared.	The cache line is read from RAM by Direct-access, and the IC_TAG, IC_XTAG, and IC_DATA registers are 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 not updated. 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.

## 20.2.6 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

**Figure 20-5 IC\_TAG Register for Reads**



**Figure 20-6 IC\_TAG Register for Writes**



Following are the definitions of the various fields in this register:

- SET\_LSB =  $\log_2(\text{IC_BSIZE})$
- SET\_MSB =  $\log_2(\text{IC_SIZE}/\text{IC_WAYS}) - 1$
- TAG\_LSB =  $\log_2(\text{IC_SIZE}/\text{IC_WAYS})$
- TAG\_MSB = 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.



### 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])

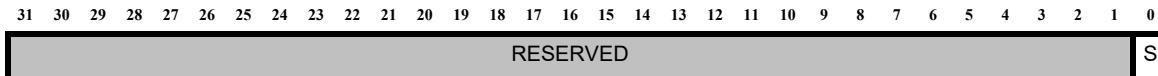
The lock bit (LCK) indicates whether the data associated with the tag is locked. The data entry is locked when LCK=1.

## 20.2.7 Instruction Cache Secure Bit Tag Register, IC\_XTAG

Address: 0x1C

Access: RW in the secure mode only

**Figure 20-7 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.

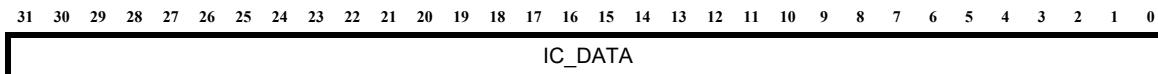
Attempts to access this register in the normal mode raises a privilege violation exception. If the secure debugger is unlocked in the normal mode, the debugger can only read the tags that are tagged in the normal mode; attempts to write to this register in the are ignored.

## 20.2.8 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

**Figure 20-8 IC\_DATA Register**



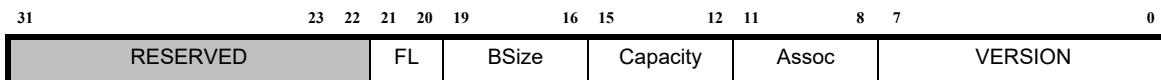
Reading from or writing to the IC\_DATA register allows the programmer to access a data word from a particular cache line, using the address in IC\_RAM\_ADDR. An SR to the IC\_RAM\_ADDRESS also triggers an update of the IC\_TAG and IC\_DATA registers.

## 20.3 Instruction Cache Configuration Register, I\_CACHE\_BUILD

Address: 0x77

Access: R

**Figure 20-9 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 20-9 I\_CACHE\_BUILD Field Descriptions**

Field Name	Bit	Description
VERSION	[7:0]	<p>Version number</p> <ul style="list-style-type: none"> <li>■ 0x0: No I_CACHE_BUILD register</li> <li>■ 0x1: Reserved</li> <li>■ 0x2: ARCompact V1, fixed 32-byte line size</li> <li>■ 0x3: ARCompact V1, variable line size</li> <li>■ 0x4: ARCV2</li> <li>■ 0x5: Added support for the secure bit in the cache tags.</li> </ul> <p><i>All other values are Reserved</i></p>
Assoc	[11:8]	<p>Cache Associativity</p> <ul style="list-style-type: none"> <li>■ 0000: Direct-mapped (1-way set associative)</li> <li>■ 0001: Two-way set associative</li> <li>■ 0010: Four-way set associative</li> <li>■ 0011: Eight-way set associative</li> </ul> <p><i>All other values are Reserved</i></p>
Capacity	[15:12]	<p>Cache capacity</p> <ul style="list-style-type: none"> <li>■ 0000: Reserved</li> <li>■ 0010: 2 Kbytes</li> <li>■ 0011: 4 Kbytes</li> <li>■ 0100: 8 Kbytes</li> <li>■ 0101: 16 Kbytes</li> <li>■ 0110: 32 Kbytes</li> </ul> <p><i>All other values are Reserved</i></p>

**Table 20-9 I\_CACHE\_BUILD Field Descriptions (Continued)**

Field Name	Bit	Description
BSize	[19:16]	<p>Block Size, indicates the cache block size in bytes.</p> <ul style="list-style-type: none"> <li>■ 0000: 8 bytes</li> <li>■ 0001: 16 bytes</li> <li>■ 0010: 32 bytes</li> <li>■ 0011: 64 bytes</li> <li>■ 0101: 256 bytes</li> </ul> <p><i>All other values are Reserved</i></p>
FL	[21:20]	<p>Feature Level, indicates locking and debug feature level</p> <ul style="list-style-type: none"> <li>■ 00: Basic cache, with cache invalidation but no locking or debug features</li> <li>■ 01: Line lock and invalidate features also supported</li> <li>■ 10: Line lock, invalidate, and advanced debug features also supported</li> <li>■ 11: Reserved</li> </ul>

## 20.4 Exceptions

See [Table 4-7](#) on page [226](#).

# 21

# DCCM

## 21.1 DCCM Introduction

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 21-1 DCCM Auxiliary Registers**

Address	Auxiliary Register Name	Description
0x18	<a href="#">DCCM Base Address, AUX_DCCM</a>	Start address of the DCCM
0x74	<a href="#">DCCM Configuration Register, DCCM_BUILD</a>	DCCM build configuration register

## 21.2 DCCM Base Address, AUX\_DCCM

Address: 0x18

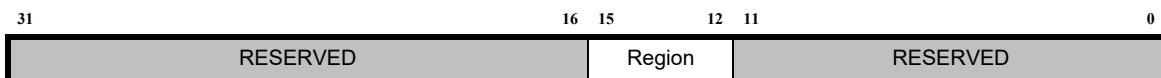
Access: RW

Default 0x8

**Figure 21-1 AUX\_DCCM, base address for DCCM (ADDR\_SIZE == 32)**



**Figure 21-2 AUX\_DCCM, base address for DCCM (ADDR\_SIZE == 16)**



**Figure 21-3 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 is reserved (RAZ/IOW). 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.

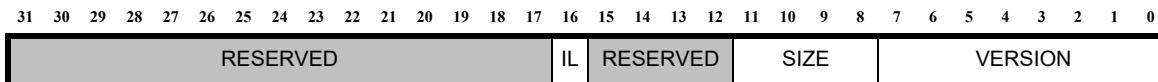
### 21.2.1 DCCM Configuration Register, DCCM\_BUILD

Address: 0x74

Access: R

The DCCM RAM Configuration register (DCCM\_BUILD) describes the size and version number of the Data Closely Coupled Memory.

**Figure 21-4 DCCM\_BUILD Register**



**Table 21-2 DCCM\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	Current version 0x5; supports configuring DCCM into secure and normal regions.
SIZE0	[11:8]	Size of DCCM RAM <ul style="list-style-type: none"> <li>■ 0x0: Not present</li> <li>■ 0x1: 512B</li> <li>■ 0x2: 1KB</li> <li>■ 0x3: 2KB</li> <li>■ 0x4: 4KB</li> <li>■ 0x5: 8KB</li> <li>■ 0x6: 16KB</li> <li>■ 0x7: 32KB</li> <li>■ 0x8: 64KB</li> <li>■ 0x9: 128KB</li> <li>■ 0xA: 256KB</li> <li>■ 0xB: 512KB</li> <li>■ 0xC: 1MB</li> <li>■ 0xD = 2 MB</li> </ul>
IL	[16]	Indicates if DCCM supports interleaving <ul style="list-style-type: none"> <li>■ 1: DCCM supports interleaving; DCCM is split into even and odd instances to enable single cycle unaligned accesses</li> <li>■ 0: DCCM does not support interleaving</li> </ul>

## 21.2.2 Exceptions

See [Table 4-7](#) on page [226](#).

# 22

## 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.

### 22.1 Data Cache Auxiliary Registers

**Table 22-1 Basic Data Cache Auxiliary Registers (DC\_FEATURE\_LEVEL==0)**

Address	Auxiliary Register Name	Description
0x47	<a href="#">Invalidate Data Cache, DC_IVDC</a>	Invalidate cache
0x48	<a href="#">Data Cache Control Register, DC_CTRL</a>	Data cache control register
0x4B	<a href="#">Flush Data Cache, DC_FLSH</a>	Flush data cache
0x5D	<a href="#">Non-cached Memory Region, AUX_CACHE_LIMIT</a>	Start address of the first non-cached data region

**Table 22-2 Data Cache Lock/Flush Registers (DC\_FEATURE\_LEVEL > 0)**

Address	Auxiliary Register Name	Description
0x49	<a href="#">Lock Data Cache Line, DC_LDL</a>	Lock data line
0x4A	<a href="#">Invalidate Data Line, DC_IVDL</a>	Invalidate data line
0x4C	<a href="#">Flush Data Line, DC_FLDL</a>	Flush data line

**Table 22-3 Data Cache Auxiliary Registers for Direct RAM Access (DC\_FEATURE\_LEVEL >1)**

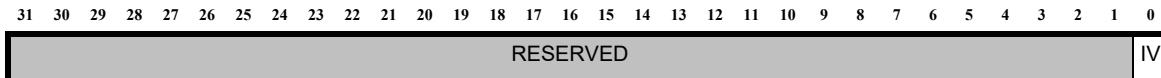
Address	Register Name	Description	Initial Value
0x58	Data Cache External Access Address, DC_RAM_ADDR	Data cache external access RAM address	0x0000 0000
0x59	Data Cache Tag Access, DC_TAG	Data cache tag access	0x0000 0000
0x5A	Data Cache Secure Bit Register, DC_XTAG	Indicates the secure mode of a cache line in the data cache tag	
0x5B	Data Cache Data Access, DC_DATA	Data cache data access	Undefined

## 22.1.1 Invalidate Data Cache, DC\_IVDC

Address: 0x47

Access: W

**Figure 22-1 DC\_IVDC Register**



DC\_IVDC is typically used to reset the data-cache contents upon a software process switch.

**Table 22-4 DC\_IVDC Field Descriptions**

Flag Name	Bit	Description	Access Type
IV	[0]	Invalidate Data Cache: Invalidates entire data cache 0 – No Action 1 – Invalidate Data Cache	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.

## When SecureShield 2+2 mode is configured

**Table 22-5 DC\_IVDC Description When SecureShield 2+2 Option is Configured**

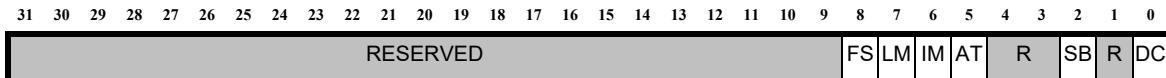
Operating Mode	Behavior
Core or debugger is in the normal mode	<ul style="list-style-type: none"> <li>■ When DC_CTRL.IM==0, the invalidate entire cache operation is ignored.</li> <li>■ When DC_CTRL.IM==1, the normal (non-secure) data cache lines are invalidated and flushed.</li> </ul>
Core or debugger is in the secure mode	<p>Writing a 1 to the IV flag (bit 0) in the DC_IVDC register invalidates and unlocks the entire data cache.</p> <p>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).</p> <p>All cache lines are invalidated and unlocked, and each line entry is processed according to the state of the IM flag.</p> <p>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.</p> <ul style="list-style-type: none"> <li>■ IM = 0 (Invalidate Only) — Any write to the DC_IVDC register invalidates all cache-line entries.</li> <li>■ 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.</li> </ul>

## 22.1.2 Data Cache Control Register, DC\_CTRL

Address: 0x48

Access: RW

**Figure 22-2 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 22-6 DC\_CTRL Control Flags**

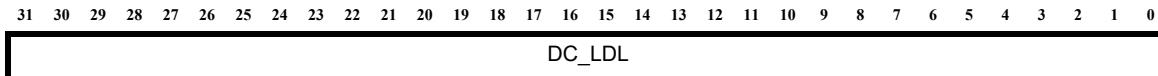
Flag Name	Bit	Description	Access Type
DC	[0]	Disable Cache: Enables/Disables the cache 0 – Enable Cache 1 – Disable Cache	R/W
SB	[2]	Success Bit: Success of last cache operation 0 – Last cache operation failed 1 – Last cache operation succeeded	R/W
AT	[5]	Address Debug Type: Used for debug purposes for when accessing cache RAMs. 0 – Direct Cache RAM Access 1 – Cache Controlled RAM Access	R/W
IM	[6]	Invalidate Mode: Selects the invalidate type 0 – Invalidate data cache only 1 – Invalidate data cache and flush dirty entries	R/W
LM	[7]	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]	Flush Status: Status of the data-cache flush mechanism 0: Idle 1 – Flush operation in progress	R

### 22.1.3 Lock Data Cache Line, DC\_LDL

Address: 0x49

Access: W

**Figure 22-3 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.
- 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.

## When SecureShield 2+2 mode option is configured

**Table 22-7 DC\_LDL Description When SecureShield 2+2 mode Option is Configured**

Operating Mode	Behavior
Core or debugger is in the normal mode	<ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> </ul>
Core or debugger is in the secure mode	<ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> </ul>

**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.

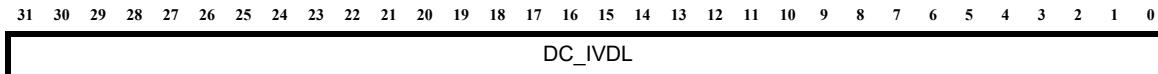
Locked cache lines can be unlocked using the `DC_IVDL` register. The `SB` flag is used to indicate a successful lock-line operation. However, if a `DC_LDL` causes a line to be flushed to memory, it is important that the `FS` flag in `DC_CTRL` be checked along with the `SB` flag to ensure that memory is coherent. Perform this check whenever memory coherency is an issue.

## 22.1.4 Invalidate Data Line, DC\_IVDL

Address: 0x4A

Access: W

**Figure 22-4 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).

The SB flag in DC\_CTRL can be checked to verify the completion of a DC\_IVDL operation.

### 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), 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).

## When SecureShield 2+2 mode option is configured

**Table 22-8 DC\_IVDL Description When SecureShield 2+2 mode Option is Configured**

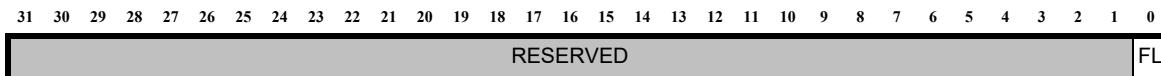
Operating Mode	Behavior
Core or debugger is in the normal mode	<ul style="list-style-type: none"> <li>■ IM = 0 (Invalidate Only) — If the system memory address written to DC_IVDL is present in the cache and the cache line is marked as present in the normal memory space based on the MPU lookup, the cache 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.</li> <li>■ 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.</li> </ul>
Core or debugger is in the secure mode	<p>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.</p> <p>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).</p> <ul style="list-style-type: none"> <li>■ 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).</li> <li>■ 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).</li> </ul>

## 22.1.5 Flush Data Cache, DC\_FLSH

Address: 0x4B

Access: W

**Figure 22-5 DC\_FLSH Register**



**Table 22-9 DC\_FLSH Control Flags**

Flag Name	Bit	Description	Access Type
FL	[0]	Flush Data Cache:- Flush entire data cache 0 – No Action 1 – Flush Data Cache	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 are 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 22-10 DC\_FLSH Description When ESP is Configured**

Operating Mode	Behavior
Core or debugger is in the normal mode	<ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> </ul>
Core or debugger is in the secure mode	<p>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 are flushed to the memory subsystem. Upon completion of a flush sequence the line-dirty bit is cleared.</p> <p>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.</p> <ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> </ul>



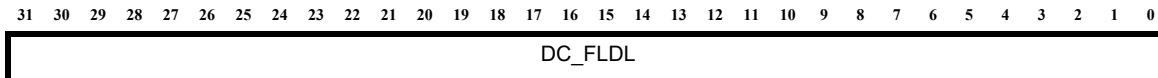
**Note** Writing a 0 to the FL flag does not initiate a flush sequence.

## 22.1.6 Flush Data Line, DC\_FLDL

Address: 0x4C

Access: W

**Figure 22-6 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). 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).

## When SecureShield 2+2 mode option is configured

**Table 22-11 DC\_FLDL Description When SecureShield 2+2 mode is Configured**

Operating Mode	Behavior
Core or debugger is in the normal mode	<ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> </ul>
Core or debugger is in the secure mode	<p>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.</p> <ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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).</li> </ul>



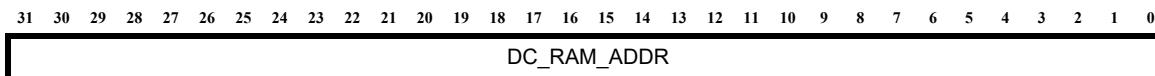
**Note** If an entry is not dirty but is present in the cache, the SB flag indicates a success (SB=1).

## 22.1.7 Data Cache External Access Address, DC\_RAM\_ADDR

Address: 0x58

Access: When ESP is not configured: RW  
When ESP is configured: RW in the secure mode and  
secure debug unlocked mode

**Figure 22-7 DC\_RAM\_ADDR**



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 22-12 Cache Line Access in the Debug Unlocked in the Normal Mode**

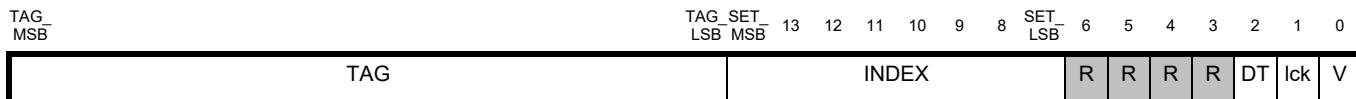
	Cache Line is in Secure Memory Space	Cache Line is in the Normal Memory Space
AT==0; and the cache line is a hit	The DC_TAG, DC_XTAG, and DC_DATA registers are not updated. The Success bit is cleared.	The cache line is read from RAM by Direct-access, and the DC_TAG, DC_XTAG, and DC_DATA registers are updated. The Success bit is set.
AT==1; and the cache line is a hit	The DC_TAG, DC_XTAG, and DC_DATA registers are not updated. 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.

## 22.1.8 Data Cache Tag Access, DC\_TAG

Address: 0x59

Access: When ESP is not configured: RW  
When ESP is configured: RW in the secure mode only

**Figure 22-8 DC\_TAG for Reads**



**Figure 22-9 DC\_TAG for Writes**



Following are the definitions of the various fields in this register:

- SET\_LSB = Log2(DC\_BSIZE)
- SET\_MSB = Log2(DC\_SIZE/DC\_WAYS) - 1
- TAG\_LSB = Log2(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 22-13 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 <ul style="list-style-type: none"> <li>■ 0 – Cache line not valid</li> <li>■ 1 – Cache line valid</li> </ul>	Programmer updates to DC_TAG DC_IVDL command DC_IVDC command Cache line replacement	R/W
LCK	[1]	Lock Flag: Lock flag associated with the cache line <ul style="list-style-type: none"> <li>■ 0 – Cache line not locked</li> <li>■ 1 – Cache line locked</li> </ul>	Programmer updates to DC_TAG DC_LDL command DC_IVDL command DC_IVDC command	R/W

**Table 22-13 DC\_TAG Control Flags and Update Conditions (Continued)**

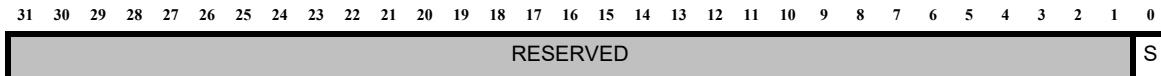
DT	[2]	Dirty Flag: Dirty flag associated with the cache line ■ 0 – Cache line not dirty ■ 1 – Cache line dirty	Programmer updates to DC_TAG DC_FLDL command DC_FLSH command DC_IVDL command DC_IVDC command Cache line replacement	R/W
----	-----	---	--	-----

## 22.1.9 Data Cache Secure Bit Register, DC\_XTAG

Address: 0x5A

Access: RW in the secure mode only

**Figure 22-10 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.

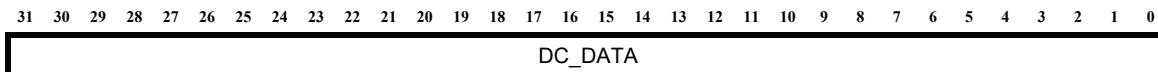
Attempts to access this register in the normal mode raises a privilege violation exception. If the secure debugger is unlocked in the normal mode, the debugger can only read the tags that are tagged in the normal mode; attempts to write to this register are ignored.

## 22.1.10 Data Cache Data Access, DC\_DATA

Address: 0x5B

Access: When ESP is not configured: RW  
When ESP is configured: RW in the secure mode only

**Figure 22-11 DC\_DATA**



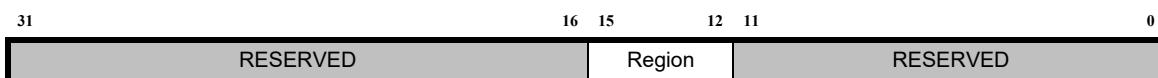
Reading from or writing to the DC\_DATA register allows the programmer to access a data from a particular cache line, using the address in DC\_RAM\_ADDR. In direct mode, an SR to this register updates the register and has the side-effect of writing the new value to the cache data RAM at the location 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.

## 22.1.11 Non-cached Memory Region, AUX\_CACHE\_LIMIT

**Figure 22-12 AUX\_CACHE\_LIMIT, start of non-cached regions (ADDR\_SIZE == 32)**



**Figure 22-13 AUX\_CACHE\_LIMIT, (ADDR\_SIZE == 16)**



**Figure 22-14 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.

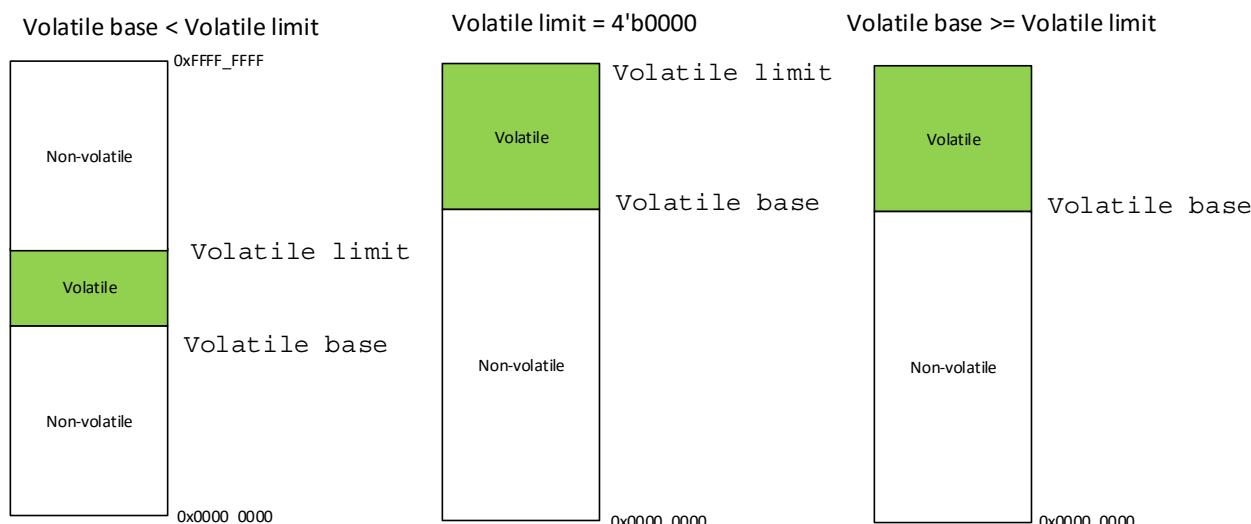
In symmetric multi-processor implementations, this register is not replicated in each core. Therefore writes to this register affect the behavior of memory operations issued from all cores equally.

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.

**Figure 22-15 Volatile Space**

## 22.2 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 22-14 Build Configuration Registers**

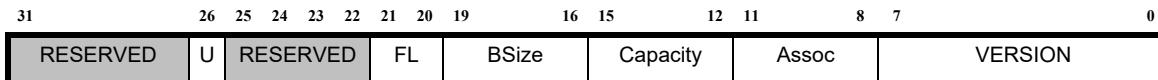
Number	Name	Access	Description
0x72	<a href="#">Data Cache Configuration Register, D_CACHE_BUILD</a>	R	Build configuration for: data cache

## 22.2.1 Data Cache Configuration Register, D\_CACHE\_BUILD

Address: 0x72

Access: R

**Figure 22-16 D\_CACHE\_BUILD**



**Table 22-15 D\_CACHE\_BUILD Field Descriptions**

Field Name	Bit	Description
VERSION	[7:0]	<p>Version number</p> <ul style="list-style-type: none"> <li>■ 0x0: No D_CACHE_BUILD register</li> <li>■ 0x1: reserved</li> <li>■ 0x2: ARCompact, fixed 32-byte line size</li> <li>■ 0x3: ARCompact, variable line size</li> <li>■ 0x4: ARCv2 with fixed number of cycles</li> <li>■ 0x5: Added support for the secure bit in the cache tags.</li> </ul> <p><i>All other values are reserved</i></p>
Assoc	[11:8]	<p>Cache Associativity</p> <ul style="list-style-type: none"> <li>■ 000: Direct-mapped (1-way set associative)</li> <li>■ 001: Two-way set associative</li> <li>■ 010: Four-way set associative</li> <li>■ 011: Eight-way set associative</li> </ul> <p><i>All other values are reserved</i></p>
Capacity	[15:12]	<p>Cache capacity</p> <ul style="list-style-type: none"> <li>■ 000: Reserved</li> <li>■ 0010: 2 Kbytes</li> <li>■ 0011: 4 Kbytes</li> <li>■ 0100: 8 Kbytes</li> <li>■ 0101: 16 Kbytes</li> <li>■ 0110: 32 Kbytes</li> </ul> <p><i>All other values are reserved</i></p>

**Table 22-15 D\_CACHE\_BUILD Field Descriptions**

Field Name	Bit	Description
BSize	[19:16]	<p>Block Size, indicates the cache block size in bytes</p> <ul style="list-style-type: none"> <li>■ 0000: 16 bytes</li> <li>■ 0001: 32 bytes</li> <li>■ 0010: 64 bytes</li> <li>■ 0100: 256 bytes</li> </ul> <p><i>All other values are reserved</i></p>
FL	[21:20]	<p>Feature Level, indicates locking and debug feature level</p> <ul style="list-style-type: none"> <li>■ 00: Basic cache, supports cache flush and invalidation operations, but no locking or debug features</li> <li>■ 01: Lock, flush, and invalidation features are supported</li> <li>■ 10: Lock, flush, invalidation, and advanced debug features are supported</li> <li>■ 11: <i>reserved</i></li> </ul>
U	26	<p>Indicates whether the data cache contains uncached regions.</p> <ul style="list-style-type: none"> <li>■ 0: Data cache does not include uncached regions.</li> <li>■ 1: Data cache includes uncached regions; the AUX_CACHE_LIMIT register is present and identifies the region at which the uncached memory regions begin.</li> </ul>

## 22.2.2 Exceptions

See [Table 4-7](#) on page [226](#).



# 23

## 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 NVM program storage using the following auxiliary registers:

### 23.1 NV\_ICCM Register Interface

Use the auxiliary registers in [Table 23-1](#) to program and control the non-volatile ICCM.

**Table 23-1 NV\_ICCM Registers**

Address	Name	Description
0x354	NVM ICCM Control Register, <a href="#">NV_ICCM_CTRL</a>	NV_ICCM control register.
0x355	NVM ICCM Erase Register, <a href="#">NV_ICCM_ERASE</a>	NV_ICCM erase register
0x356	NVM ICCM Programming Register, <a href="#">NV_ICCM_PROG</a>	NV_ICCM programming register
0x357	NVM ICCM Data Register, <a href="#">NV_ICCM_DATA</a>	NV_ICCM data register
0x78	ICCM Configuration Register, <a href="#">ICCM_BUILD</a>	Build Configuration Register

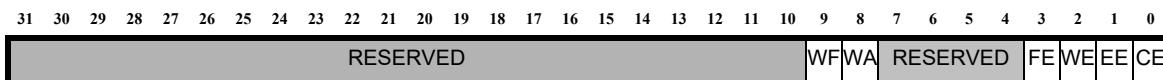
### **23.1.1 NVM ICCM Control Register, NV\_ICCM\_CTRL**

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 23-1 NV\_ICCM\_CTRL Register**



Use this register to program the non-volatile ICCM (NV\_ICCM).

**Table 23-2 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 WE bit to permit write or erase operations on NV_ICCM. If the WE bit is set to 0, and you try to program or erase NV_ICCM using auxiliary registers, the WF bit is set to 1, and the program or erase operation is not executed.
FE	[3]	<p>Program failure exception enable. When this bit is set to 1, the following events raise exceptions:</p> <ul style="list-style-type: none"> <li>■ Programming an address that is out of range of the NV_ICCM region.</li> <li>■ Program failures during the NV_ICCM write and erase operations.</li> </ul> <p>When this bit is set to 0, these events do not raise exceptions, but set the NV_ICCM_CTRL.WF bit to 1.</p>
WA	[8]	When the WA bit is set to 1, this bit indicates that a program or erase operation is in progress. Writes to the WA bit are ignored.

**Table 23-2 NV\_ICCM\_CTRL Field Descriptions (Continued)**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
WF	[9]	<p>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 NV_ICCM_CTRL.FE==0:</p> <ul style="list-style-type: none"> <li>■ Programming an address that is out of range of the NV_ICCM region.</li> <li>■ Program failures during NVM write and erase operations.</li> </ul> <p>This bit is also set to 1, when NV_ICCM_CTRL.WE==0 and you attempt to program or erase NV_ICCM using the programming auxiliary register: NV_ICCM_PROG.</p>

### 23.1.2 NVM 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 23-2 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==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 23-3 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.

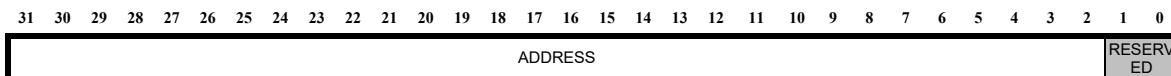
### 23.1.3 NVM ICCM Programming Register, NV\_ICCM\_PROG

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 23-3 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 erase 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.WA==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.

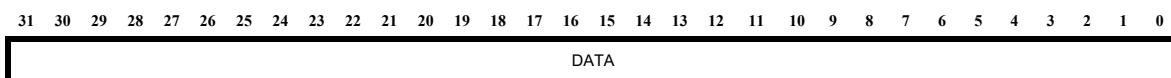
### 23.1.4 NVM ICCM Data Register, NV\_ICCM\_DATA

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 23-4 NV\_ICCM\_DATA Register**



This register specifies the data to be written into NV\_ICCM.

## 23.2 Exceptions

See [Table 4-7](#) on page [226](#).

# 24

## Cryptographic Key Storage

### 24.1 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.

### 24.2 Cryptographic Key Storage Register Interface

**Table 24-1 Calibration Storage Registers**

Address	Name	Description
0x350	Key Store Control Register, KEY_STORE_CTRL	Key storage control register.
0x351	Key Store Password Register, KEY_STORE_PWD	Key storage password register
0x352	Key Store Address Register, KEY_STORE_ADDR	Key storage address register
0x353	Key Store Data Register, KEY_STORE_DATA	Key storage data register
0x35C	Key Store SID Register, KEY_STORE_SID	Key storage SID register.

**Table 24-1 Calibration Storage Registers**

Address	Name	Description
0xD8	Key Store Build Configuration Register, KEY_STORE_BUILD	Key storage build configuration register

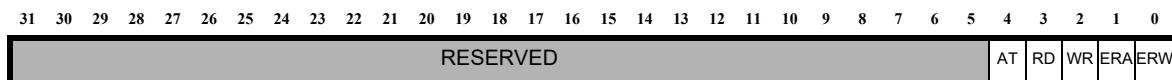
## 24.2.1 Key Store Control Register, KEY\_STORE\_CTRL

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 24-1 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 24-2 KEY\_STORE\_CTRL Field Descriptions**

Field	Bit	Description
ERW	[0]	<p>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:</p> <ol style="list-style-type: none"> <li>1. Ensure that the <code>KEY_STORE_CTRL.AT</code> bit is set to 0.</li> <li>2. Write the destination address in the <code>KEY_STORE_ADDR</code> register.</li> <li>3. To trigger the operation, write the password (default: 0x5A) to the <code>KEY_STORE_PWD</code> register.</li> </ol> <p>This bit is write-only. Reads of this bit return 0.</p> <p>Note: If you do not write the password to the <code>KEY_STORE_PWD</code> register, the operation is ignored.</p>
ERA	[1]	<p>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:</p> <ol style="list-style-type: none"> <li>1. Ensure that the <code>KEY_STORE_CTRL.AT</code> bit is set to 0.</li> <li>2. Write the destination address in the <code>KEY_STORE_ADDR</code> register.</li> <li>3. To trigger the operation, write the password (default: 0x5A) to the <code>KEY_STORE_PWD</code> register.</li> </ol> <p>This bit is write-only. Reads of this bit return 0.</p> <p>Note: If you do not write the password to the <code>KEY_STORE_PWD</code> register, the operation is ignored.</p>

**Table 24-2 KEY\_STORE\_CTRL Field Descriptions**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
WR	[2]	<p>Write enable. Write a 1 to this bit to trigger a write operation to the key storage. Before enabling this bit, do the following:</p> <ol style="list-style-type: none"> <li>1. Ensure that the KEY_STORE_CTRL.AT bit is set to 0.</li> <li>2. Write the destination address in the KEY_STORE_ADDR register.</li> <li>3. Write the data to the KEY_STORE_DATA register.</li> <li>4. To trigger the operation, write the password (default: 0x5A) to the KEY_STORE_PWD register.</li> </ol> <p>This bit is write-only. Reads of this bit return 0.</p> <p>Note: If you do not write the password to the KEY_STORE_PWD register, the operation is ignored.</p>
RD	[3]	<p>Read enable. Write a 1 to this bit to trigger a read operation from the key storage. Before enabling this bit, do the following:</p> <ol style="list-style-type: none"> <li>1. Ensure that the KEY_STORE_CTRL.AT bit is set to 0.</li> <li>2. Write the source address, from where the data is read, in the KEY_STORE_ADDR register.</li> </ol> <p>After the read operation, the data is returned in the KEY_STORE_DATA register.</p> <p>This bit is write-only. Reads of this bit return 0.</p>
AT	[4]	<p>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.</p>

 **Note**

- When the ERW, ERA, WR, or RD operations are programmed, no further operations are triggered if the NVM 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.

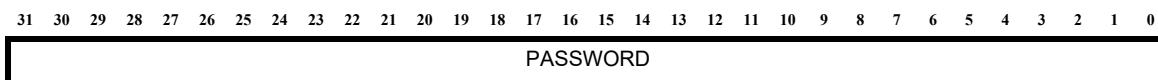
## 24.2.2 Key Store Password Register, KEY\_STORE\_PWD

Address: 0x351

- Access:
- When SecureShield 2+2 mode is not configured: W
  - When SecureShield 2+2 mode is configured: W in the secure mode only

Reset 0x0

**Figure 24-2 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 24-3 KEY\_STORE\_PWD Field Descriptions**

Field	Bit	Description
PASSWORD	[31:0]	<p>Password that grants you privileges to program and erase the key storage.</p> <p>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.</p>

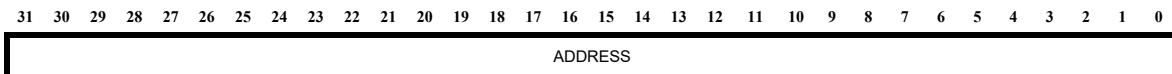
### 24.2.3 Key Store Address Register, KEY\_STORE\_ADDR

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 24-3 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.

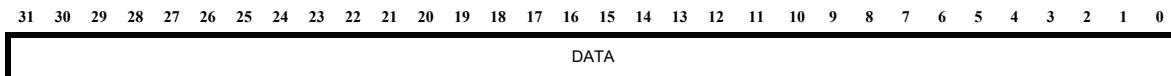
## 24.2.4 Key Store Data Register, KEY\_STORE\_DATA

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 24-4 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.

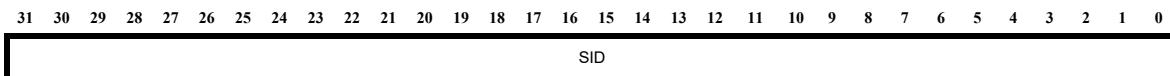
## 24.2.5 Key Store SID Register, KEY\_STORE\_SID

Address: 0x35C

Access: RW in the secure mode

Reset: 0x00

**Figure 24-5 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.

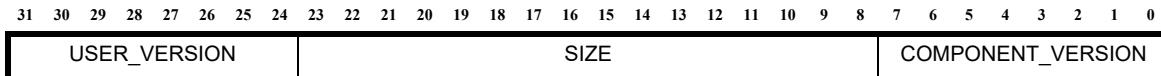
This register is accessible only in the secure kernel mode. Attempts to access this register from the secure user mode results in a basic privilege violation (0x070000). Attempts to access this register from a normal mode triggers a EV\_PrivilegeV exception with ECR 0x071020.

## 24.2.6 Key Store Build Configuration Register, KEY\_STORE\_BUILD

Address: 0xD8

Access: R

**Figure 24-6 KEY\_STORE\_BUILD Register**



This register contains the cryptographic key storage configuration.

**Table 24-4 KEY\_STORE\_BUILD**

Field	Bit	Description
COMPONENT_VERSION	[7:0]	Version of the crypto 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 crypto key storage component, configurable in ARChitect



# 25

## 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 25-1 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

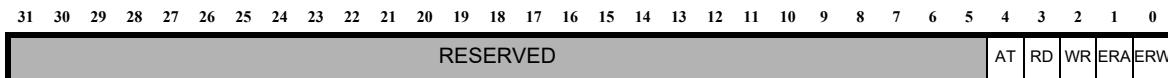
## 25.0.1 Calibration Parameter Store Control Register, CAL\_STORE\_CTRL

Address: 0x358

Access: RW

Reset: 0x0

**Figure 25-1 CAL\_STORE\_CTRL Register**



Use this register to read, write, and erase the calibration parameter storage non-volatile memory.

**Table 25-2 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 CAL_STORE_CTRL.AT bit is set to 0. 2. Write the destination address in the CAL_STORE_ADDR 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 CAL_STORE_CTRL.AT bit is set to 0. 2. Write the destination address in the CAL_STORE_ADDR register. This bit is write-only. Reads to 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 CAL_STORE_CTRL.AT bit is set to 0. 2. Write the destination address in the CAL_STORE_ADDR register. 3. Write the data to the CAL_STORE_DATA register. This bit is write-only. Reads to this bit return 0.

**Table 25-2 CAL\_STORE\_CTRL Field Descriptions**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
RD	[3]	<p>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:</p> <ol style="list-style-type: none"> <li>1. Ensure that the CAL_STORE_CTRL.AT bit is set to 0.</li> <li>2. Write the source address, from where the data is read, in the CAL_STORE_ADDR register.</li> </ol> <p>After the read operation, the data is returned in the CAL_STORE_DATA register.</p> <p>This bit is write-only. Reads of this bit return 0.</p>
AT	[4]	<p>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.</p>

 **Note**

- When the ERW, ERA, WR, or RD operations are programmed, no further operations are triggered if the NVM 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.

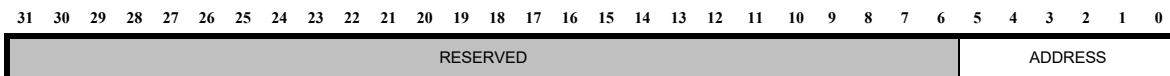
## 25.0.2 Calibration Parameter Store Address Register, CAL\_STORE\_ADDR

Address: 0x359

Access: RW

Reset: 0x0

**Figure 25-2 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.

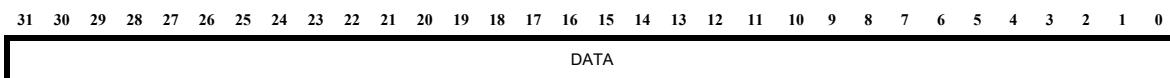
### 25.0.3 Calibration Parameter Store Data Register, CAL\_STORE\_DATA

Address: 0x35A

Access: RW

Reset: 0x00

**Figure 25-3 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.

## 25.1 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.

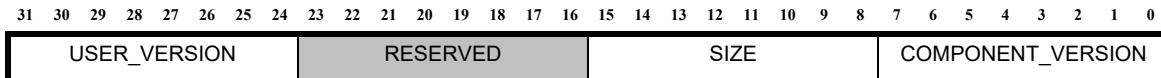
Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCv2-based system.

### 25.1.1 Calibration Parameter Store Build Configuration Register, CAL\_STORE\_BUILD

Address: 0xD9

Access: R

**Figure 25-4 CAL\_STORE\_BUILD Register**



This register contains the calibration parameter storage configuration.

**Table 25-3 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

# 26

# Processor Timers

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. The default interrupt priority level of timer 0 is set to P1 and default interrupt priority of timer 1 is set to P0.
- 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. The default interrupt priority level of ST0 is set to P1 and default interrupt priority of ST1 is set to P0.
- a watchdog timer to detect time-based errors as opposed to non-time-based errors (a random bit flip in memory). The module is used to detect these errors, accurately providing a way to initiate a system reset or interrupt in response, thereby invoking a recovery process.

## 26.1 Timers

The processor timers are connected to a system clock signal that operates even when the ARCV2-based processor is in the sleep state. The timers can be used to generate interrupt signals that wake the processor from the SLEEP state. For more information about internal clocks during the sleep state, see [Table 29-3](#) on page [1051](#).

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.

### 26.1.0.1 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.

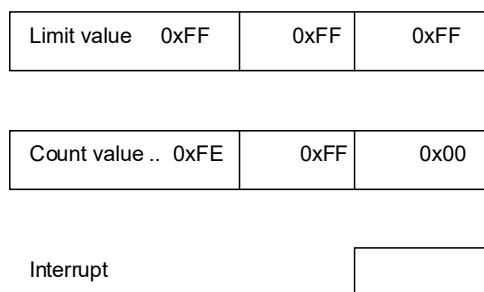
- Set up the control flags according to the desired mode of operation by updating the timer `CONTROL $n$`  register.
- Write the count value to the timer `count $n$`  register.

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 26-1 Interrupt Generated after Timer Reaches Limit Value**



The software must clear the timer interrupts. 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. In Watchdog mode, the reset signal is activated two cycles after the limit condition is reached.

## 26.2 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 26-1](#) and [Table 26-2](#).

**Table 26-1 Auxiliary Registers for Hardware Counter 0 (has\_timer\_0 == true)**

Address	Auxiliary Register Name	Description
0x21	<a href="#">Timer 0 Count Register, COUNT0</a>	Processor timer 0 count value
0x22	<a href="#">Timer 0 Control Register, CONTROL0</a>	Processor timer 0 control value
0x23	<a href="#">Timer 0 Limit Register, LIMIT0</a>	Processor timer 0 limit value

**Table 26-2 Auxiliary Registers for Hardware Counter 1 (has\_timer\_1 == true)**

Address	Auxiliary Register Name	Description
0x100	<a href="#">Timer 1 Count Register, COUNT1</a>	Processor timer 1 count value

**Table 26-2 Auxiliary Registers for Hardware Counter 1 (has\_timer\_1 == true) (Continued)**

Address	Auxiliary Register Name	Description
0x101	Timer 1 Control Register, CONTROL1	Processor timer 1 control value
0x102	Timer 1 Limit Register, LIMIT1	Processor timer 1 limit value

**Table 26-3 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 26-4 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 26-5 Auxiliary Registers for Hardware Counter 0 (-sec\_timer\_1==true)**

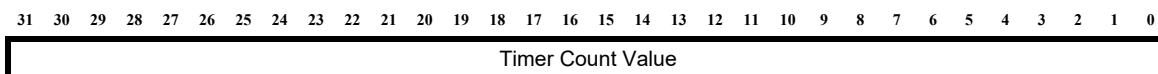
Address	Auxiliary Register Name	Description
0x109	Secure Timer 1 Count Register, AUX_ST1_COUNT	Processor timer 0 count value
0x10A	Secure Timer 1 Control Register, AUX_ST1_CTRL	Processor timer 0 control value
0x10B	Secure Timer 1 Limit Register, AUX_ST1_LIMIT	Processor timer 0 limit value

### 26.2.1 Timer 0 Count Register, COUNT0

Address: 0x21

Access: RW

**Figure 26-2 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.

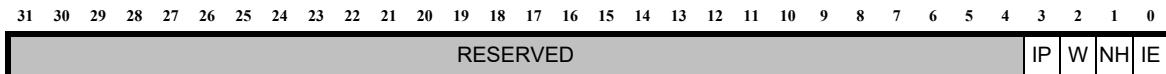
This register contains 0x00000000 on reset.

## 26.2.2 Timer 0 Control Register, CONTROL0

Address: 0x22

Access: RW

**Figure 26-3 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 start the next iteration of counting, setting COUNT0 to 0, when LIMIT0 equals COUNT0.

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.

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.

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.

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. If both the IE and W bits are cleared, 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.

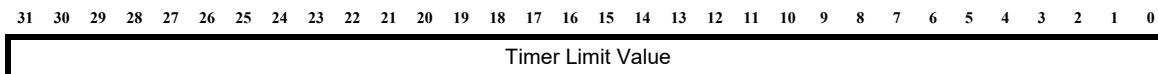
The IP bit is set when the COUNTn register reaches the LIMITn value, and remains set until cleared by the timer interrupt service routine. The IP bit can be cleared by re-writing the desired values of W, NH and IE into the CONTROL0 register, thereby writing a 0 into the IP position. An interrupt service 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.

### 26.2.3 Timer 0 Limit Register, LIMIT0

Address: 0x23

Access: RW

**Figure 26-4 Timer 0 Limit Value Register**



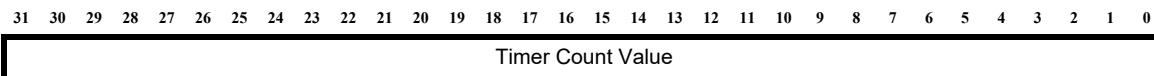
You must write the limit value into this register. The limit value is the value after which an interrupt or a reset must be generated. For backward compatibility to previous processor variants, the timer limit register is set to 0x00FFFFFF when the processor is [Reset](#).

## 26.2.4 Timer 1 Count Register, COUNT1

Address: 0x100

Access: RW

**Figure 26-5 Timer 1 Count Value Register**



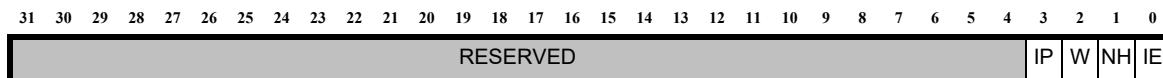
See [Timer 0 Count Register, COUNT0](#) on page 1012 for field information.

## 26.2.5 Timer 1 Control Register, CONTROL1

Address: 0x101

Access: RW

**Figure 26-6 Timer 1 Control Register**



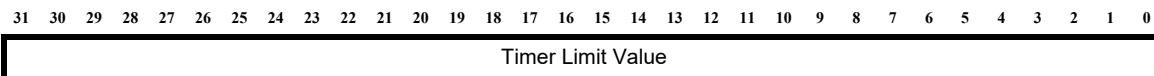
See [Timer 0 Control Register, CONTROL0](#) on page 1013 for field information.

## 26.2.6 Timer 1 Limit Register, LIMIT1

Address: 0x102

Access: RW

**Figure 26-7 Timer 1 Limit Value Register**



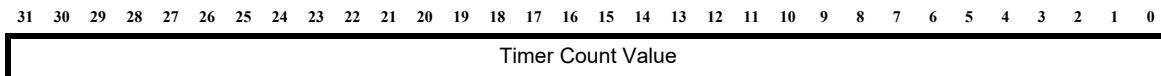
See [Timer 0 Limit Register, LIMIT0](#) on page 1014 for field information.

### 26.2.7 Secure Timer 0 Count Register, AUX\_ST0\_COUNT

Address: 0x106

Access: RW in the secure mode only

**Figure 26-8 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.

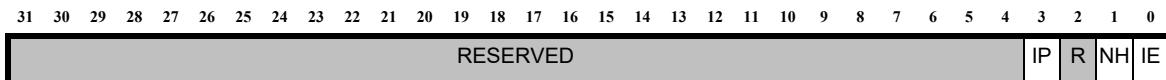
This register contains 0x00000000 on reset.

## 26.2.8 Secure Timer 0 Control Register, AUX\_ST0\_CTRL

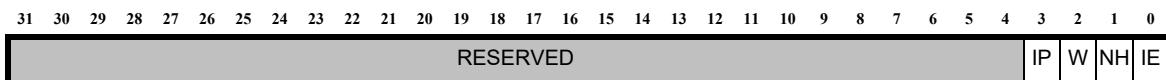
Address: 0x107

Access: RW in the secure mode only

**Figure 26-9 Secure Timer 0 Control Register**



**Figure 26-10 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.

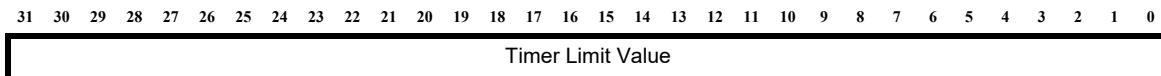
The IP bit is set when the secure timer count register reaches the secure timer limit value, and remains set until cleared by the secure timer interrupt service routine. The IP bit can be cleared by re-writing the desired values of NH and IE into the secure timer control register, thereby writing a 0 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.

## 26.2.9 Secure Timer 0 Limit Register, AUX\_ST0\_LIMIT

Address: 0x108

Access: RW in the secure mode only

**Figure 26-11 Timer 0 Limit Value Register**



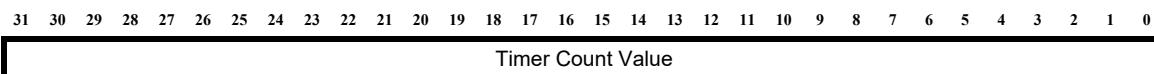
You must write the limit value into this register. The limit value is the value after which an interrupt or a reset must be generated. The timer limit register is set to 0x00FFFFFF when the processor is [Reset](#).

### 26.2.10 Secure Timer 1 Count Register, AUX\_ST1\_COUNT

Address: 0x109

Access: RW in the secure mode only

**Figure 26-12 Timer 1 Count Value Register**



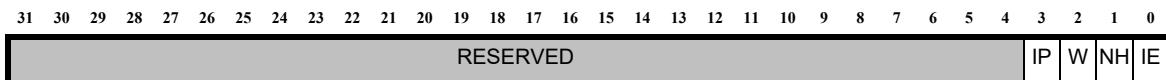
See “Secure Timer 0 Count Register, AUX\_ST0\_COUNT” on page [1018](#) for field information.

### 26.2.11 Secure Timer 1 Control Register, AUX\_ST1\_CTRL

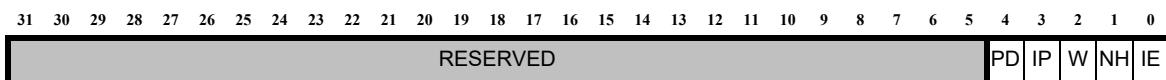
Address: 0x10A

Access: RW in the secure mode only

**Figure 26-13 Timer 1 Control Register**



**Figure 26-14 Timer 1 Control Register**



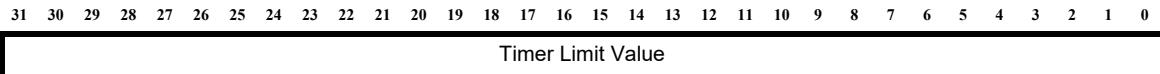
See “[Secure Timer 0 Control Register, AUX\\_ST0\\_CTRL](#)” on page [1019](#)for field information.

### 26.2.12 Secure Timer 1 Limit Register, AUX\_ST1\_LIMIT

Address: 0x10B

Access: RW in the secure mode only

**Figure 26-15 Timer 1 Limit Value Register**



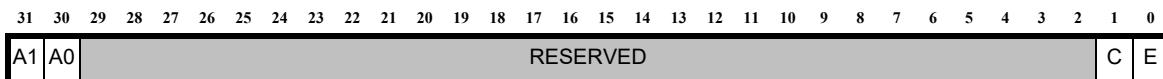
See “Secure Timer 0 Limit Register, AUX\_ST0\_LIMIT” on page 1020 for field information.

### 26.2.13 RTC Control Register, AUX\_RTC\_CTRL

Address: 0x103

Access: rW

**Figure 26-16 AUX\_RTC\_CTRL Register**

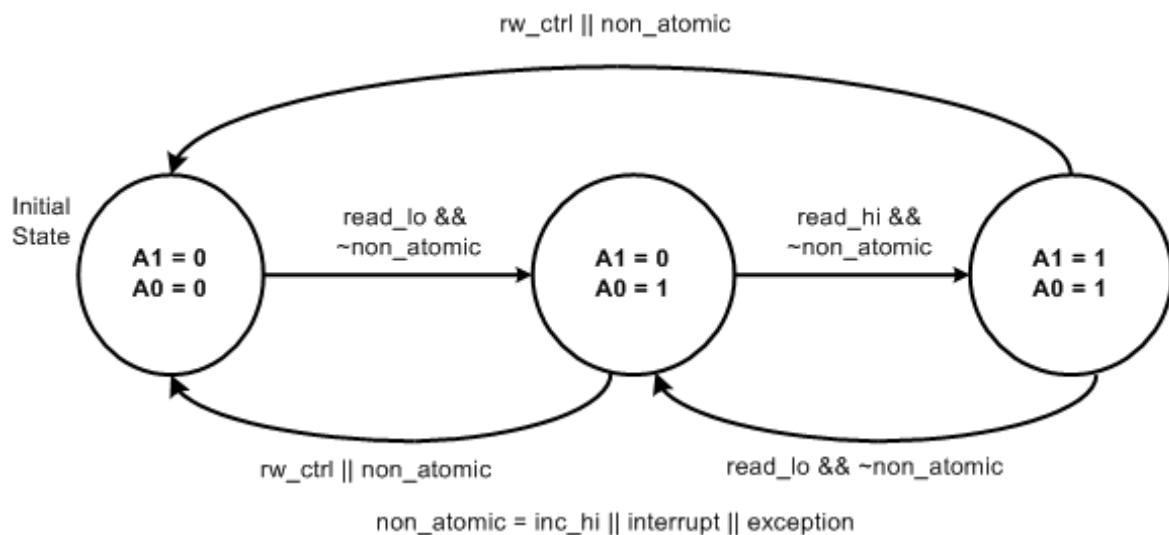


The 64-bit RTC is a free-running counter. This counter does not have any associated interrupts. The RTC is controlled using the AUX\_RTC\_CTRL register. The following are the field descriptions:

**Table 26-6 RTC Control Register**

Bits	Bit	Description
E	[0]	Enable A value of 0 means disabled; 1 means enable counting.
C	[1]	A value of 1 clears the AUX_RTC_LOW and AUX_RTC_HIGH registers.
A1, A0	[31:30]	These bits track the atomicity of reads from the <a href="#">RTC Count High Register, AUX_RTC_HIGH</a> and <a href="#">RTC Count Low Register, AUX_RTC_LOW</a> registers. For the description of the state machine, see <a href="#">Figure 26-17</a> on page <a href="#">1025</a> .

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 26-17 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).

### 26.2.13.1 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.



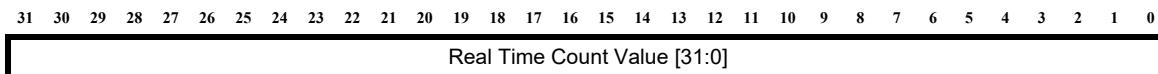
**Note** Placing the reads inside a critical region ensures atomicity, but does not detect when the AUX\_RTC\_LOW register wraps around between the two reads. Hence, always use the above scheme.

### 26.2.14 RTC Count Low Register, AUX\_RTC\_LOW

Address: 0x104

Access: r

**Figure 26-18 RTC Count Low Register**



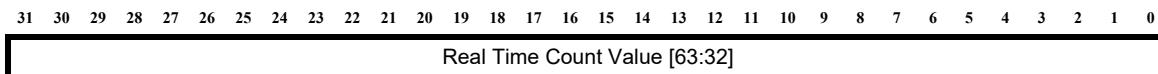
The RTC count low register is a read register in user mode. Reading this register returns the LSB 32-bits of the free-running RTC.

### 26.2.15 RTC Count High Register, AUX\_RTC\_HIGH

Address: 0x105

Access: r

**Figure 26-19 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.

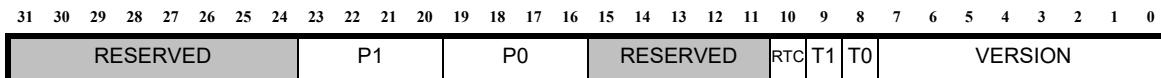
## 26.2.16 Timers Configuration Register, TIMER\_BUILD

Address: 0x75

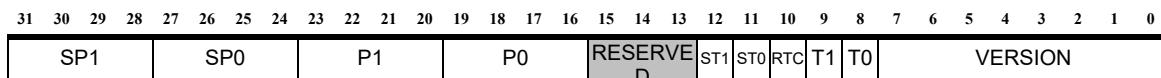
Access: R

The TIMER\_BUILD configuration register (TIMER\_BUILD) indicates the presence of the processor timers auxiliary registers.

**Figure 26-20 TIMER\_BUILD Register**



**Figure 26-21 TIMER\_BUILD Register**



**Table 26-7 TIMER\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Current version</b> <ul style="list-style-type: none"> <li>■ 0x01: Version 1</li> <li>■ 0x02: ARC 700 Processor Timers</li> <li>■ 0x03: ARC 600 R3 Processor Timers, with interrupt-pending bits</li> <li>■ 0x04: ARCv2 Processor Timers</li> <li>■ 0x05: ARCv2 with secure Timers</li> </ul>
T0	[8]	<b>Timer 0 Present</b> <ul style="list-style-type: none"> <li>■ 0x0: No timer 0</li> <li>■ 0x1: Timer 0 present</li> </ul>
T1	[9]	<b>Timer 1 Present</b> <ul style="list-style-type: none"> <li>■ 0x0: No timer 1</li> <li>■ 0x1: Timer 1 present</li> </ul>
RTC	[10]	<b>64-bit RTC Configuration</b> <ul style="list-style-type: none"> <li>■ 0x0: 64-bit RTC is disabled.</li> <li>■ 0x1: 64-bit RTC is enabled.</li> </ul>
ST0	[11]	<b>Secure timer 0</b> <ul style="list-style-type: none"> <li>■ 0x0: No secure timer 0</li> <li>■ 0x1: Secure timer 0 is present</li> </ul>

**Table 26-7 TIMER\_BUILD Field Descriptions (Continued)**

Field	Bit	Description
ST1	[12]	<b>Secure timer 1</b> <ul style="list-style-type: none"> <li>■ 0x0: No secure timer 1</li> <li>■ 0x1: Secure timer 1 is present</li> </ul>
P0	[19:16]	<b>Indicates the interrupt priority level of Timer 0</b> Note: If Timer 0 is not included, this field is always set to 0.
P1	[23:20]	<b>Indicates the interrupt priority level of Timer 1</b> Note: If Timer 1 is not included, this field is always set to 0.
SP0	[27:24]	<b>Indicates the interrupt priority level of Secure Timer 0</b> Default: priority level 1
SP1	[31:28]	<b>Indicates the interrupt priority level of Secure Timer 1</b> Default: priority level 0

## 26.3 Watchdog Timers

The ARCV2 watchdog timer module is used to detect time-based errors as opposed to non-time-based errors (a random bit flip in memory). The module is used to detect these errors, accurately providing a way to initiate a system reset or interrupt in response, thereby invoking a recovery process.

A common time-based error is a system deadlock. This is where some unexpected random event has caused the system to hang. This could be hardware inflicted, for example, a bus interface lock 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-relevant systems. The watchdog timer has the following features:

- Ability to monitor a period of time with the option to:
  - Generate an interrupt in response to a timed-out event (irq 18)
  - Signal system modules in response to a timed-out event
  - Automatically initiate a system reset in response to a timed-out event

**Table 26-8 Error Protection Auxiliary Registers**

Address	Auxiliary Register Name	Description
0x238	<a href="#">WDT_PASSWD</a>	Watchdog password register
0x239	<a href="#">WDT_CTRL</a>	Watchdog timer control register
0x23A	<a href="#">WDT_PERIOD</a>	Watchdog timer period register
0x23B	<a href="#">WDT_COUNT</a>	Watchdog timer count register

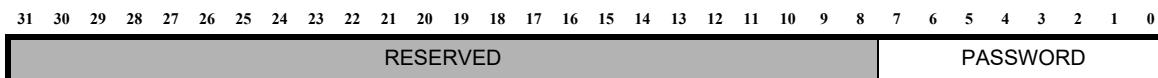
## 26.4 Watchdog Password Register, WDT\_PASSWD

Address: 0x238

- Access:
- When SecureShield 2+2 mode is not configured: W
  - When SecureShield 2+2 mode is configured: W in the secure mode only

Reset 0x00

**Figure 26-22 WDT\_PASSWD Register**



The watchdog password register is used to access the control and period registers. Three passwords are defined. Two of the passwords unlocks the respective watchdog timer register for write access.

Exceptions are raised when you attempt to do the following operations on this register:

- Writes to this register in user mode result in a Privilege Violation exception.
- Reads from this register in kernel or user mode result in an Illegal Instruction exception.
- Access to this register in the secure user mode result in a Privilege Violation exception (0x070000).
- Access to this register in non-secure kernel or user mode result in an EV\_PrivilegeV exception with ECR: 0x071020.

**Table 26-9 WDT\_PASSWD Field Descriptions**

Field	Bit	Description
PASSWORD	[7:0]	Password to access the watchdog timer auxiliary registers. <ul style="list-style-type: none"> <li>■ 0xAA: Password to unlock WDT_CTRL for write access.</li> <li>■ 0x55: Password to unlock WDT_PERIOD for write access.</li> <li>■ 0x5A: Password that lets you automatically load the value in the WDT_PERIOD register to the watchdog timer counter.</li> </ul>

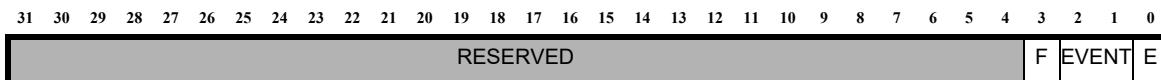
## 26.5 Watchdog Timer Register, WDT\_CTRL

Address: 0x239

- 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 26-23 WDT\_CTRL Register**



The watchdog control register is used to configure the timer. The register is four-bits in width and is broken down into the following fields.

Attempts to read and write to the register in user mode result in a Privilege Violation exception.

Access to this register in the secure user mode result in a Privilege Violation exception (0x070000).

Access to this register in non-secure kernel or user mode result in an EV\_PrivilegeV exception with ECR: 0x071020.

**Table 26-10 WDT\_CTRL Field Descriptions**

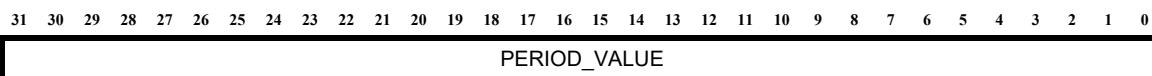
Field	Bit	Description
E	[0]	Enable watchdog timer
EVENT	[2:1]	Specifies the action the watchdog timer takes when the counter reaches zero: <ul style="list-style-type: none"> <li>■ 00: Assert (active high) external signal wdt_timedout_r.</li> <li>■ 01: Raise an interrupt (irq number 18).</li> <li>■ 10: Assert (active high) external reset enable signal wdt_reset.</li> </ul>
F	[3]	Indicates if a time out has occurred. <ul style="list-style-type: none"> <li>■ 1: the watchdog internal counter has reached the value 0 (timed out). Can be written to clear the flag.</li> <li>■ 0: For an interrupt configured event, writing a 0 to this bit clears the interrupt.</li> </ul>

## 26.6 Watchdog Timer Period Register, WDT\_PERIOD

Address: 0x23A

- Access:
- When SecureShield 2+2 mode is not configured:  
RW
  - When SecureShield 2+2 mode is configured: RW  
in the secure mode only

**Figure 26-24 WDT\_PERIOD Register**



This register stores the start value of the watchdog-timer counter. This register can store a 32-bit period value 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.

Attempts to read and write to the register in user mode result in a Privilege Violation exception.

Access to this register in the secure user mode result in a Privilege Violation exception (0x070000).

Access to this register in non-secure kernel or user mode result in an EV\_PrivilegeV exception with ECR: 0x071020.

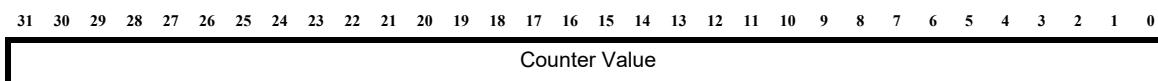
## 26.7 Watchdog Timer Count Register, WDT\_COUNT

Address: 0x23B

- Access:
- When SecureShield 2+2 mode is not configured: R
  - When SecureShield 2+2 mode is configured: R in the secure mode only

Reset 0x00

**Figure 26-25 WDT\_COUNT Register**



The watchdog timer count register holds the current value of the internal counter. The register is read only in kernel mode. A privilege violation exception is raised attempting to read the register in user mode. Attempting to write to the register in either kernel or user mode generates an illegal instruction exception.

Access to this register in the secure user mode result in a Privilege Violation exception (0x070000).

Access to this register in non-secure kernel or user mode result in an EV\_PrivilegeV exception with ECR: 0x071020.



# 27

## Safety Island

### 27.1 Safety Island Overview

The safety island configuration includes a main core and a shadow core running in lockstep. The two processors are initialized to the same state during system start-up, and both the cores receive the same inputs at their external interfaces. The safety island architecture run the same code on both the cores. The outputs from both the cores are compared each cycle, and any mismatches are flagged. The safety island architecture does not indicate in which core the error has occurred. Some failures, such as radiation, can cause identical failures in both the cores. To mitigate the occurrence of such failures, the shadow core is brought out of reset a few cycles after the main core. As both the cores receive the same inputs, a delay is added to the inputs of the shadow core. For cycle by cycle comparisons to work, the outputs of the main core are delayed by the same number of cycles as the input delay for the shadow core. You must implement an external logic that monitors the error signaling from the lockstep cores and performs corrective actions as dictated by the system. Corrective action is not taken by either the main core or the shadow core.



**Note** You can also run the two cores in a dual-core mode where the lockstep controller logic is disabled, and the core run independently of each other.

The safety island architecture involves the following logic:

- The two cores are configured with the lockstep controller:
  - The main core executes the target application and the shadow verifies the main core operation.
  - Outputs from the shadow core are used for checking only; the outputs of the main core only are used to communicate externally

The lockstep controller includes the following components:

- Core Safety Monitor (Comparator block) is the comparator module that is responsible for detecting mismatches between the two processor core outputs, recording fault information and signaling failures.
- Interface Handler: Additional logic that is used to connect the two cores in an effective lockstep configuration. An example is the synchronization of asynchronous inputs.

For more information about the lockstep signals and the safety island implementation, see the *ARC EM Series Bookshelf*.

## 27.2 Dual-Core Lockstep Initialization After Reset

You must perform the lockstep initialization after the following events:

- The core is reset: You can configure the delay cycles between the main core and the shadow core when building the processor. The lockstep controller delays the reset, halt, and run signals to the two cores based on the configured delay cycles. Hence, after a core is reset, the initialization sequence must be performed to maintain the delay between the cores.
- Delay loop buffers are locked due to mismatches or errors.

Following is the initialization sequence

1. Execute a simple loop for 10 cycles to flush the core registers.
2. The main core resets delay buffer lock by writing to the LSC\_MCD[2] bit.
3. Each core writes 1 to the LSC\_CTRL[0] bit requesting the lockstep controller to enable lockstep for both the cores.
4. The lockstep controller writes 1 to the LSC\_CTRL[1] bit if both the cores are enabled in lockstep. The lockstep controller must write to the LSC\_CTRL[1] bit within three cycles of receiving the enable request from the cores. Otherwise, the core times out. Hence, poll the LSC\_CTRL[1] bit to verify that the lockstep is enabled.
5. Run your code normally.

The lockstep controller is initialized and starts monitoring the cores for mismatches.

### Example 27-1 Sample Lockstep Initialization Sequence

```
.text
.global _start
    ; Start of test - above the vector table
_start:
    ; Lockstep Initialization
    ;
    mov r0, 0x0
    mov r28, stack_area
    mov lp_count, 10
    lp    loop_end
        nop
loop_end:
    mov r1, 4; Reset lock
    sr r1, [0xa81]
    nop    ; Enable Lockstep
    mov r1, 1
    sr r1, [0xa80]
    nop
    nop
    mov r2, 3
poll:   lr r0, [0xa80]
    brne r0, r2, poll
        lr r0, [0xa82]; Done
    nop
```

## 27.3 Halting and Running Cores that are in Lockstep

To halt a core, write 1 to the LSC\_MCD [0] bit.

To run a core, write 1 to the LSC\_MCD [1] bit.

## 27.4 Safety Island Auxiliary Register Interface

**Table 27-1 Safety Island Registers**

Address	Name	Description
0xA80	“Lockstep Control Register, LSC_CTRL”	Lockstep enable register
0xA81	“Lockstep Debug Control Register, LSC_MCD”	Lockstep: lock reset, run, and halt register
0xA82	“Lockstep Status Register, LSC_STATUS”	Lockstep: mismatch and lock status register
0xA83	“Lockstep Error Register, LSC_ERROR_REG”	Lockstep error register
0xEF	“Lockstep Configuration Register, LSC_BUILD_AUX”	Lockstep build configuration register

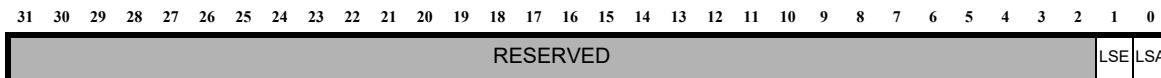
### 27.4.1 Lockstep Control Register, LSC\_CTRL

Address: 0xA80

Access: RW

Default: 0x0

**Figure 27-1 LSC\_CTRL Register**



Use this register to request the lockstep controller to enable lockstep with the other core. After the lockstep controller acknowledges this request, the lockstep controller updates the `LSC_CTRL.LSE` bit to indicate whether the lockstep with the other core has been enabled or not.

For a dual-core lockstep, the lockstep controller enables lockstep based on the `LSC_CTRL.LSA` bits of both the core as follows:

**Table 27-2 Enabling Lockstep Modes**

Core 0 LSC_CTRL.LSA	Core 1 LSC_CTRL.LSA	Lockstep Mode
0	0	Disabled
0	1	Disabled
1	0	Disabled
1	1	Enabled

**Table 27-3 LSC\_CTRL Register**

Field	Bit	Description
LSA	[0]	<p>Activate lockstep request to the lockstep controller</p> <ul style="list-style-type: none"> <li>■ 0: Disable lockstep; clear this bit to send a disable lockstep request. It takes up to two cycles for the lockstep to disable error propagation.</li> <li>■ 1: Enable lockstep; set this bit to 1 to send a lockstep enable request.</li> </ul> <p>Note: For two cores to be in lockstep, the <code>LSC_CTRL.LSA</code> bits in both cores must be set to 1.</p>

**Table 27-3 LSC\_CTRL Register**

Field	Bit	Description
LSE	[1]	<p>Lockstep is enabled</p> <ul style="list-style-type: none"><li>■ 0: Lockstep is disabled by the lockstep controller.</li><li>■ 1: Lockstep controller has acknowledged the lockstep enable request, and the lockstep is enabled for the core.</li></ul> <p>Note: This bit must be updated by the lockstep controller within three clock cycles of the <code>LSC_CTRL.LSA</code> bit is updated. Otherwise, the core times out.</p>

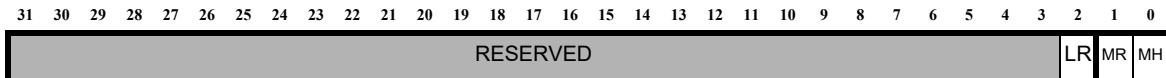
## 27.4.2 Lockstep Debug Control Register, LSC\_MCD

Address: 0xA81

Access RW

Reset 0x0

**Figure 27-2 LSC\_MCD Register**



Use this register to halt and run a core running in a lockstep with another core. You can also reset the locks on the delay buffers.

**Table 27-4 LSC\_MCD Field Description**

Field	Bits	Description
MH (MCD_HALT)	[0]	Write a 1 to this bit to force halt the processor.
MR (MCD_RUN)	[1]	Write a 1 to this bit to run the processor from a halt state.
LR (Lock Reset)	[2]	Write a 1 to this register to reset the lock on the delay buffers. When there is a mismatch between the cores, the delay buffers are locked.

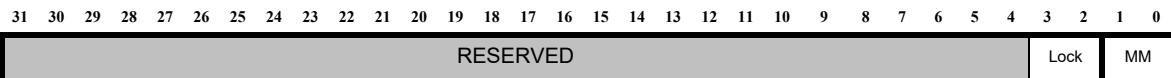
### 27.4.3 Lockstep Status Register, LSC\_STATUS

Address: 0xA82

Access: R

Access: 0x12

**Figure 27-3 LSC\_STATUS**



Use this register to read the status of the lock condition of the delay buffers and mismatch status. This register is updated on a read request.

**Table 27-5 LSC\_STATUS Field Description**

Field	Bits	Description
MM	[1:0]	<ul style="list-style-type: none"> <li>■ 0x0:Indicates a mismatch</li> <li>■ 0x1:Indicates a match</li> <li>■ 0x2: Indicates a match</li> <li>■ 0x3: Indicates a mismatch</li> </ul>
Lock	[3:2]	<ul style="list-style-type: none"> <li>■ 0x0:reserved</li> <li>■ 0x1:Indicates the delay buffers are unlocked.</li> <li>■ 0x2: Indicates the delay buffers are locked.</li> <li>■ 0x3: reserved</li> </ul>

## 27.4.4 Lockstep Error Register, LSC\_ERROR\_REG

Address: 0xA83

Access: R

Access: 0x0

**Figure 27-4 LSC\_ERROR\_REG**



**Table 27-6 LSC\_ERROR\_REG Field Description**

Field	Bits	Description
Lockstep	[0]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the last instruction committed.</li> <li>■ 0x0: No mismatch on any DMA channels.</li> </ul>
	[1]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the architectural PC.</li> <li>■ 0x0: No mismatch on any DMA channels.</li> </ul>
	[2]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the DMP write data.</li> <li>■ 0x0: No mismatch on the DMP write data.</li> </ul>
	[3]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the DMP write address.</li> <li>■ 0x0: No mismatch on the DMP write address.</li> </ul>
	[4]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the register file0 write data.</li> <li>■ 0x0: No mismatch on the register file0 write data.</li> </ul>
	[5]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the register file1 write data.</li> <li>■ 0x0: No mismatch on the register file1 write data.</li> </ul>
	[6]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the merged register file write address.</li> <li>■ 0x0: No mismatch on the merged register file write adders.</li> </ul>
	[7]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch in the pipe state bits.</li> <li>■ 0x0: No mismatch in the pipe state bits.</li> </ul>
	[8]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the ECR register.</li> <li>■ 0x0: No mismatch on the ECR register.</li> </ul>
	[9]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the STATUS32 register.</li> <li>■ 0x0: No mismatch on the STATUS32 register.</li> </ul>

**Table 27-6 LSC\_ERROR\_REG Field Description**

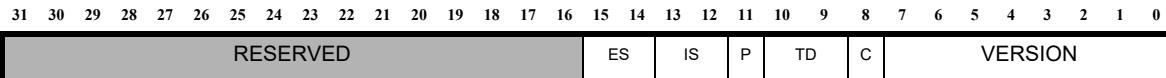
Field	Bits	Description
DMA	[17]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on a DMA channels.</li> <li>■ 0x0: No mismatch on any DMA channels.</li> </ul>
ECC	[18]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the main core and the shadow core ECC error signals.</li> <li>■ 0x0: No mismatch on the main core and the shadow core ECC error signals.</li> </ul>
Parity	[19]	<ul style="list-style-type: none"> <li>■ 0x1: Indicates a parity error on the delay buffers.</li> <li>■ 0x0: No parity error on the delay buffers.</li> </ul> <p>This bit is valid only if parity checking is configured on the delay buffers.</p>
uDMA	[20]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the DMA interface.</li> <li>■ 0x0: No mismatch on the DMA interface.</li> </ul>
WDT	[21]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the watchdog timer interface.</li> <li>■ 0x0: No mismatch on the watchdog timer interface.</li> </ul>
DMI	[22]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the DMI DCCM interface.</li> <li>■ 0x0: No mismatch on the DMI DCCM interface.</li> </ul>
	[23]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the DMI ICCM interface.</li> <li>■ 0x0: No mismatch on the DMI ICCM interface.</li> </ul>
UAUX	[24]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the UAFX interface.</li> <li>■ 0x0: No mismatch on the UAFX interface.</li> </ul>
PER	[25]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the peripheral interface.</li> <li>■ 0x0: No mismatch on the peripheral interface.</li> </ul>
DBU	[26]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the DBU interface.</li> <li>■ 0x0: No mismatch on the DBU interface.</li> </ul>
IBU	[27]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on the IBU AHB interface.</li> <li>■ 0x0: No mismatch on the IBU AHB interface.</li> </ul>
SYS	[28]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on system control unit such as halt and sleep.</li> <li>■ 0x0: No mismatch on system control unit.</li> </ul>
EXT	[29]	<ul style="list-style-type: none"> <li>■ 0x1: indicates error on user configured external logic such as APEX.</li> <li>■ 0x0: No error on the user configured external logic.</li> </ul>
ICCM	[30]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on ICCM.</li> <li>■ 0x0: No mismatch on ICCM.</li> </ul>
DCCM	[31]	<ul style="list-style-type: none"> <li>■ 0x1: Mismatch on DCCM.</li> <li>■ 0x0: No mismatch on DCCMs.</li> </ul>

## 27.4.5 Lockstep Configuration Register, LSC\_BUILD\_AUX

Address: 0xEF

Access: R

**Figure 27-5 LSC\_BUILD\_AUX**



Read this register to determine the lockstep controller configuration in the processor.

**Table 27-7 LSC\_BUILD\_AUX Field Description**

Field	Bit	Description
VERSION	[7:0]	<ul style="list-style-type: none"> <li>Set to non-zero if lockstep controller is present in the build.</li> </ul>
C	[8]	<ul style="list-style-type: none"> <li>Indicates if the main and the shadow core share CCMs or they use separate CCMs</li> <li>0x0: Separate CCM</li> <li>0x1: Shared CCM</li> </ul>
TD	[10:9]	<ul style="list-style-type: none"> <li>Indicates the number of delay cycles between the cores; the shadow core lags the main core.</li> <li>0x00 – Zero Delay</li> <li>0x01 – 1 Cycle Delay</li> <li>0x10 or 0x11 – 2 Cycle Delay</li> </ul>
P	[11]	<ul style="list-style-type: none"> <li>Indicates parity protection for the delay buffers</li> <li>0x0: No parity protection</li> <li>0x1: Parity protection is supported</li> </ul>
IS	[13:12]	<ul style="list-style-type: none"> <li>Indicates the number of synchronization stages for asynchronous inputs such as halt, reset, and run.</li> <li>0x01: One synchronization stage</li> <li>0x02: Two synchronization stages</li> </ul>
ES	[15:14]	<ul style="list-style-type: none"> <li>Indicates the number of synchronization stages for asynchronous external interrupts</li> <li>0x01: One synchronization stage</li> <li>0x02: Two synchronization stages</li> </ul>

# 28

## 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 run time through the DMP\_PER\_AUX register.

The peripheral region is non-cacheable. The component can be added to the core in the ARChitect GUI. This component has the following option:

- -per\_bus\_reg\_interface: <true|false>

Where

true = add clocked staging register inside the component.

false = flow-through

### 28.1 DMP Auxiliary Registers

**Table 28-1 DMP Registers**

Address	Auxiliary Register Name	Description
0x20A	DMP_PER_AUX	Peripheral memory region programming register

## 28.2 Peripheral Memory Region, DMP\_PER\_AUX

Address: 0x20A

Access: ■ When SecureShield 2+2 mode is not configured: RW  
■ When SecureShield 2+2 mode is configured: RW in the secure mode only

When a peripheral memory region is configured, the DMP\_PER\_AUX register exists, and identifies the region at which the peripheral memory region begins.

On reset, the Region field is set to 15, which is the highest region in the memory map. This region is also non-cacheable, by default. To ensure a consistent view of data, the data cache must be flushed whenever DMP\_PER\_AUX is modified.



### 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

# 29

## External Host Debugging

### 29.1 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 respective ARCV2 processor databook.

### 29.2 Host Debug Register Interface

**Table 29-1 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

## 29.3 Debug Register, DEBUG

Address: 0x05

Access: RG

**Figure 29-1 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 [1059](#).

The debug register (DEBUG) contains the following bits:

**Table 29-2 DEBUG Register Field Description**

Field	Bit	Description
Force halt (FH)	[1]	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]	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 <a href="#">STATUS32</a> ) is cleared, for example, by restarting (see <a href="#">Starting</a> ) or single-stepping (see <a href="#">Single Instruction Stepping</a> ) the ARCv2-based processor. actionpoints that cause a software exception set 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 29-2 DEBUG Register Field Description**

Field	Bit	Description
Actionpoints Status Register field (ASR)	[10:3]	<p>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. When a new actionpoint or actionpoints are triggered, all other previously set ASR bits are cleared and the ASR field reflects only the newly triggered actionpoint(s).</p> <p>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 <a href="#">Actionpoint Match Value, AP_AMV0</a>, <a href="#">Actionpoint Match Mask, AP_AMM0</a>, and <a href="#">Actionpoint Control, AP_AC0</a>).</p> <p>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.</p>
Single instruction step (IS)	[11]	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.
EP	[19]	<p>This enable bit controls the gated clock driving the modules associated with the performance counters.</p> <ul style="list-style-type: none"> <li>■ 1: Enables the clock to the performance counters</li> <li>■ 0: Disables the clock to the performance counters</li> </ul>
ED	[20]	<p>This enable bit controls the gated clock driving the modules associated with the actionpoints.</p> <ul style="list-style-type: none"> <li>■ 1: Enables the clock to the actionpoints module</li> <li>■ 0: Disables the clock to the actionpoints module</li> </ul>
External Halt (EH)	[21]	<p>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 wishes 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> acknowledgement signal. This bit is a status bit, and the host cannot write to this bit.</p>

**Table 29-2 DEBUG Register Field Description**

Field	Bit	Description
Reset applied (RA)	[22]	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]	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]	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 <a href="#">Table 29-3</a> . This bit is a status bit, and the host cannot write to this bit.
Triple Fault (TF)	[27]	When this bit is set to 1, it indicates that the processor has halted due to a triple fault. This bit is exposed as an external pin from the core (dbg_tf_r). 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]	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.
Breakpoint halt (BH)	[29]	BH bit is set when execution of a breakpoint instruction is attempted. 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.
Self halt (SH)	[30]	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.
Load pending bit (LD)	[31]	The host (see <a href="#">“The Host” on page 849</a> ) or the ARCv2-based processor can read the LD bit at any time and indicate that there is an outstanding load waiting to complete. The host must wait for this bit to clear before changing the state of the ARCv2-based processor. This bit is a status bit, and the host cannot write to this bit.

Table 29-3 lists the state of various internal clocks during various sleep modes.

**Table 29-3 Internal Clock During Sleep Mode**

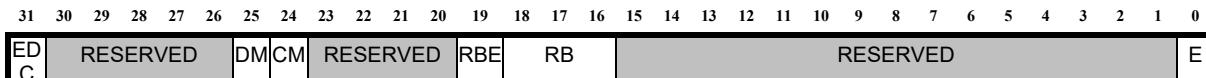
Sleep Mode (DEBUG.SM)	Core	Timers	Real-time Clock	IRQ Resync Clock
0	Disabled	Enabled	Enabled	Disabled unless required
1	Disabled	Disabled	Enabled	Disabled unless required
2	Disabled	Enabled	Disabled	Disabled unless required
3	Disabled	Disabled	Disabled	Disabled unless required
4	Disabled	Disabled	Disabled	Disabled unless required
5	Disabled	Disabled	Disabled	Disabled unless required
6	Disabled	Disabled	Disabled	Disabled unless required
7	Disabled	Disabled	Disabled	Disabled unless required

### 29.3.1 Interrupt and Register Bank Debug Register, DEBUGI

Address: 0x0F

Access: RG

**Figure 29-2 DEBUGI Register**



**Table 29-4 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.
RB	[18:16]	Register Bank (present with <code>RGF_NUM_BANKS &gt; 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, &lt; 2, &lt; 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.
CM	[24]	Indicates the operating mode of the core <ul style="list-style-type: none"> <li>■ 0x0: normal mode</li> <li>■ 0x1: secure mode</li> </ul> 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 <ul style="list-style-type: none"> <li>■ 0x0: normal mode</li> <li>■ 0x1: secure mode</li> </ul> This bit is valid only when <code>-sec_modes_option==true</code> . Otherwise, this bit is read as zero and ignored on writes.

**Table 29-4 DEBUGI Register Field Description**

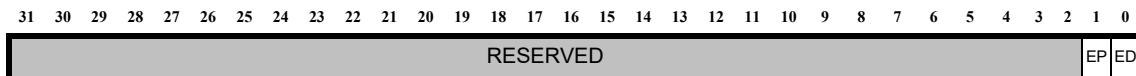
Field	Bit	Description
EDC	[31]	For ARC EM4-based processors, this bit is reserved. 1: Memory accesses by the debugger are encrypted. 0: Memory accesses by the debugger are not encrypted.

### 29.3.2 Architectural Clock Gating Control Register, ACG\_CTRL

Address: 0x30

Access: RW

**Figure 29-3 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 29-5 ACG\_CTRL Field Description**

Field	Bit	Description
ED	[0]	<p>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.</p> <p>Set this bit to 1 to enable the actionpoint clock. If this bit is set to 0, the actionpoint clock is gated.</p> <p>This bit has the same functionality as the DEBUG [20] bit, but with the extra functionality that ACG_CTRL [ED] is also accessible in kernel mode by the software.</p>
EP	[1]	<p>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.</p> <p>Set this bit to 1 to enable the performance monitor clock. If this bit is set to 0, the performance monitor clock is gated.</p> <p>This bit has the same functionality as the DEBUG [19] bit, but with the extra functionality that ACG_CTRL [EP] is also accessible in kernel mode by the software.</p>

# 30

# Debug Features

---

## 30.1 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

## 30.2 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 read data value 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.

### 30.2.1 Actionpoint Auxiliary Registers

Actionpoints are programmed using a collection of control and status registers mapped into the auxiliary register space.

**Table 30-1 Auxiliary Registers for Actionpoints**

Auxiliary Register Address	Register Name	Description
0x05	DEBUG	Extensions to Debug Register; (see <a href="#">Actionpoint Extensions to Debug Register, DEBUG, 0x05</a> )
0x76	AP_BUILD	Actionpoint build configuration register(see <a href="#">Actionpoints Configuration Register, AP_BUILD</a> )
0x220	AP_AMV0	Actionpoint 0 Match Value (see <a href="#">Actionpoint Match Value, AP_AMV0</a> )
0x221	AP_AMM0	Actionpoint 0 Match Mask (see <a href="#">Actionpoint Match Mask, AP_AMM0</a> )
0x222	AP_AC0	Actionpoint 0 Control (see <a href="#">Actionpoint Control, AP_AC0</a> )
0x223	AP_AMV1	Actionpoint 1 Match Value (see <a href="#">Actionpoint Match Value, AP_AMV1</a> )
0x224	AP_AMM1	Actionpoint 1 Match Mask (see <a href="#">Actionpoint Match Mask, AP_AMM1</a> )
0x225	AP_AC1	Actionpoint 1 Control (see <a href="#">Actionpoint Control, AP_AC1</a> )
0x226	AP_AMV2	Actionpoint 2 Match Value (see <a href="#">Actionpoint Match Value, AP_AMV2</a> )
0x227	AP_AMM2	Actionpoint 2 Match Mask (see <a href="#">Actionpoint Match Mask, AP_AMM2</a> )
0x228	AP_AC2	Actionpoint 2 Control (see <a href="#">Actionpoint Control, AP_AC2</a> )
0x229	AP_AMV3	Actionpoint 3 Match Value (see <a href="#">Actionpoint Match Value, AP_AMV3</a> )
0x22A	AP_AMM3	Actionpoint 3 Match Mask (see <a href="#">Actionpoint Match Mask, AP_AMM3</a> )
0x22B	AP_AC3	Actionpoint 3 Control (see <a href="#">Actionpoint Control, AP_AC3</a> )
0x22C	AP_AMV4	Actionpoint 4 Match Value (see <a href="#">Actionpoint Match Value, AP_AMV4</a> )
0x22D	AP_AMM4	Actionpoint 4 Match Mask (see <a href="#">Actionpoint Match Mask, AP_AMM4</a> )
0x22E	AP_AC4	Actionpoint 4 Control (see <a href="#">Actionpoint Control, AP_AC4</a> )

**Table 30-1 Auxiliary Registers for Actionpoints (Continued)**

Auxiliary Register Address	Register Name	Description
0x22F	AP_AMV5	Actionpoint 5 Match Value (see <a href="#">Actionpoint Match Value, AP_AMV5</a> )
0x230	AP_AMM5	Actionpoint 5 Match Mask (see <a href="#">Actionpoint Match Mask, AP_AMM5</a> )
0x231	AP_AC5	Actionpoint 5 Control (see <a href="#">Actionpoint Control, AP_AC5</a> )
0x232	AP_AMV6	Actionpoint 6 Match Value (see <a href="#">Actionpoint Match Value, AP_AMV6</a> )
0x233	AP_AMM6	Actionpoint 6 Match Mask (see <a href="#">Actionpoint Match Mask, AP_AMM6</a> )
0x234	AP_AC6	Actionpoint 6 Control (see <a href="#">Actionpoint Control, AP_AC6</a> )
0x235	AP_AMV7	Actionpoint 7 Match Value (see <a href="#">Actionpoint Match Value, AP_AMV7</a> )
0x236	AP_AMM7	Actionpoint 7 Match Mask (see <a href="#">Actionpoint Match Mask, AP_AMM7</a> )
0x237	AP_AC7	Actionpoint 7 Control (see <a href="#">Actionpoint Control, AP_AC7</a> )
0x23F	AP_WP_PC	Watchpoint Program Counter (see <a href="#">Watchpoint Program Counter, AP_WP_PC</a> )
0x30	ACG_CTRL	Register to enable or disable the actionpoint clocks.



For more details on the access code definitions of the registers, see [Table 3-6 – “Key to Auxiliary Register Access Permissions for LR and SR Instructions”](#) on page 106.

### 30.2.1.1 Access Requirements

All actionpoint registers are accessible only in kernel mode; accessing these registers in user mode raises a Privilege Violation exception.

Actionpoint halt in user mode requires that the DEBUG.UB bit is set.

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

### 30.2.1.2 Actionpoint Registers Host and Core Access

If both the processor core and host attempt to write to an actionpoint register at the same time, the host write always takes priority. To ensure that any write by the processor core in such situation is not lost, you must have a control-access procedure for the host and processor core.

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.

### 30.2.1.3 Actionpoint Registers Reset State

[Table 30-2](#) lists the values of the actionpoint registers when the ARCv2-based processor is reset.

**Table 30-2 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

**Table 30-2 State of Auxiliary Registers upon Reset (Continued)**

Register Name	Initial Reset Values
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
AP_WP_PC	0x00000000

### 30.2.2 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. When a new actionpoint or actionpoints are triggered, all other previously set ASR bits are cleared and the ASR field reflects only the newly triggered actionpoint(s).

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](#)).

### 30.2.3 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 1054.

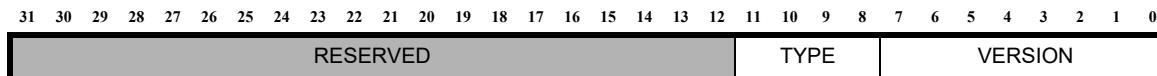
### 30.2.4 Actionpoints Configuration Register, AP\_BUILD

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 30-1 Actionpoint Build Register**



**Table 30-3 Actionpoint Build Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Actionpoint Version</b> 0x05: Current version of the actionpoint system
TYPE	[11:8]	<b>Actionpoint Build Type</b> Refers to the number of actionpoints selected in the ARCv2-based system: <ul style="list-style-type: none"> <li>■ 0000: 2 Actionpoints, full set of targets</li> <li>■ 0001: 4 Actionpoints, full set of targets</li> <li>■ 0010: 8 Actionpoints, full set of targets</li> <li>■ 0100: 2 Actionpoints, minimum set of targets</li> <li>■ 0101: 4 Actionpoints, minimum set of targets</li> <li>■ 0110: 8 Actionpoints, minimum set of targets</li> <li>■ 00011, 0111-1111: Reserved</li> </ul>

### 30.2.5 Actionpoint Match Value, AP\_AMV0

Address: 0x220

Access: RW

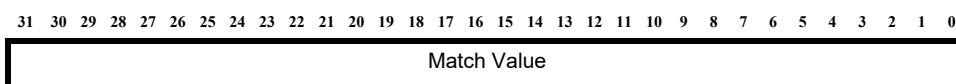
This register, in conjunction with the Actionpoint Match Mask register, specifies the condition that must be met for the associated actionpoint to trigger. This register contains the value that the monitored target must match.

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.

**Figure 30-2 Actionpoint 0 Match Value Register**



### 30.2.6 Actionpoint Match Mask, AP\_AMM0

Address: 0x221

Access: RW

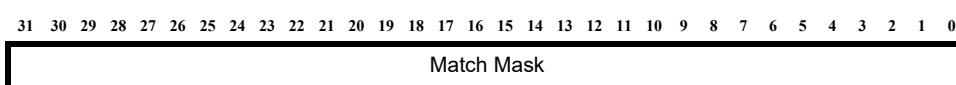
This register contains the mask that is applied to both the monitored parameter and the match value (AP\_AMV0) before the equality check is done.

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.

**Figure 30-3 Actionpoint 0 Match Mask 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.

### 30.2.7 Actionpoint Control, AP\_AC0

Address: 0x222

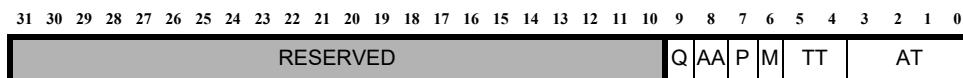
Access: RW

Sets the target and operation mode.

This register is set to 0x00000000 on reset.

This register is a read and write register.

**Figure 30-4 Actionpoint 0 Control Register**



#### 30.2.7.1 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 30-4](#) lists the selections available.

**Table 30-4 Actionpoint Target Field, AT, Type**

Actionpoint Type (AT)	Type of Data Comparison	Description
0000	Instruction address	Trigger on specified PC value.
0001	Instruction data	Trigger on specified instruction data.
0010	Memory address	Trigger on access to specific memory address, from processor core only.
0011	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.
0100	Auxiliary register address	Trigger on access to specific auxiliary register address.
0101	Auxiliary register data	Trigger on data value written to, or read from an auxiliary register from the core only.
0110	External parameter 0	Trigger on a specific value.
0111	External parameter 1	Trigger on a specific value.
1000-1111	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.

**30.2.7.2 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 30-5](#) lists the possible values: none (disabled), read, write, read or write.

**Table 30-5 Transaction Type Field, TT, Type**

Transaction Type (TT)	Description
00	Disable the associated actionpoint.
01	Trigger when the transaction is a write and the target value matches.
10	Trigger when the transaction is a read and the target value matches.
11	Trigger when the transaction is a write or read and the target value matches. If the target is memory, there is an implicit selection of the write data value.

**30.2.7.3 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 30-6](#).

**Table 30-6 Mode Field, M, Type**

Mode (M)	Description
0	Trigger when in range
1	Trigger when outside range

### 30.2.7.4 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. Hence, the two triggering events never coincide.

**Table 30-7 Pair Field, P, Type**

Pair (P)	Description
0	Normal unpaired operation
1	Pair this actionpoint with the next actionpoint

### 30.2.7.5 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.

**Table 30-8 Actionpoint Action Field, AA Field, Type**

Actionpoint Action (AA)	Description
0	<p>Halt the processor</p> <p>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.</p> <p>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.</p>
1	<p>Raise an exception</p> <p>Cause a Privilege Violation exception (see <a href="#">Privilege Violation, Action-Point Hit Instruction Fetch</a>). This exception causes the program flow to be redirected through the EV_PrivilegeV vector.</p> <p>The Cause Code and Parameter field of the ECR indicate whether an Actionpoint was triggered and which Actionpoint caused the exception.</p> <p>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.</p> <p>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.</p>

### 30.2.7.6 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  $nn$  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. Hence, at least two of the four triggering events can never coincide.

**Table 30-9 Quad Type**

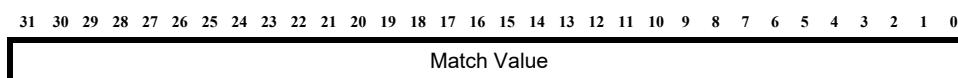
Quad (Q)	Description
0	Normal operation
1	Quad this actionpoint with the next three actionpoints (modulo the number of actionpoints)

### 30.2.8 Actionpoint Match Value, AP\_AMV1

Address: 0x223

Access: RW

**Figure 30-5 Actionpoint 1 Match Value Register**

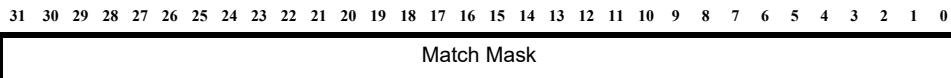


See [Actionpoint Match Value, AP\\_AMV0](#) register for field information.

### 30.2.9 Actionpoint Match Mask, AP\_AMM1

Address: 0x224

Access: RW

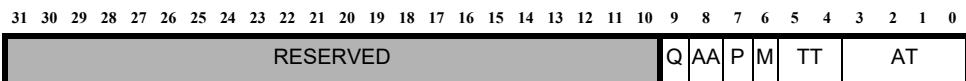
**Figure 30-6 Actionpoint 1 Match Mask Register**

See [Actionpoint Match Mask, AP\\_AMM0](#) register for field information.

### 30.2.10 Actionpoint Control, AP\_AC1

Address: 0x225

Access: RW

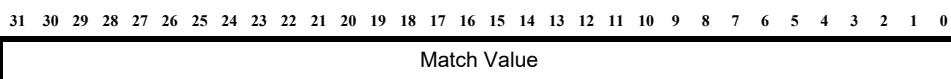
**Figure 30-7 Actionpoint 1 Control Register**

See [Actionpoint Control, AP\\_AC0](#) register for field information.

### 30.2.11 Actionpoint Match Value, AP\_AMV2

Address: 0x226

Access: RW

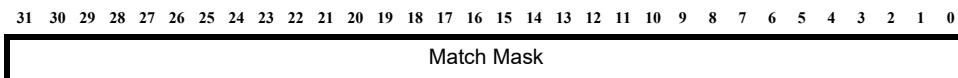
**Figure 30-8 Actionpoint 2 Match Value Register**

See [Actionpoint Match Value, AP\\_AMV0](#) register for field information.

### 30.2.12 Actionpoint Match Mask, AP\_AMM2

Address: 0x227

Access: RW

**Figure 30-9 Actionpoint 2 Match Mask Register**

See [Actionpoint Match Mask, AP\\_AMM0](#) register for field information.

### 30.2.13 Actionpoint Control, AP\_AC2

Address: 0x228

Access: RW

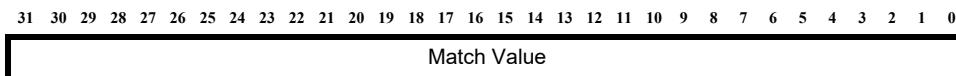
See [Actionpoint Control, AP\\_AC0](#) register for field information.

### 30.2.14 Actionpoint Match Value, AP\_AMV3

Address: 0x229

Access: RW

**Figure 30-10 Actionpoint 3 Match Value Register**



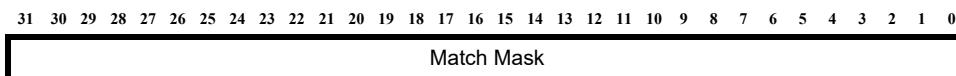
See [Actionpoint Match Value, AP\\_AMV0](#) register for field information.

### 30.2.15 Actionpoint Match Mask, AP\_AMM3

Address: 0x22A

Access: RW

**Figure 30-11 Actionpoint 3 Match Mask Register**

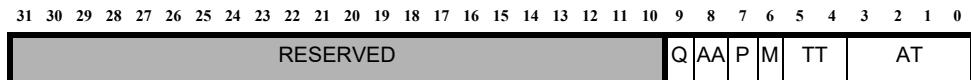


See [Actionpoint Match Mask, AP\\_AMM0](#) register for field information.

### 30.2.16 Actionpoint Control, AP\_AC3

Address: 0x22B

Access: RW

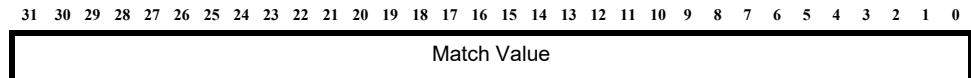
**Figure 30-12 Actionpoint 3 Control Register**

See [Actionpoint Control, AP\\_AC0](#) register for field information.

### 30.2.17 Actionpoint Match Value, AP\_AMV4

Address: 0x22C

Access: RW

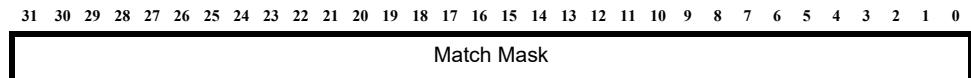
**Figure 30-13 Actionpoint 4 Match Value Register**

See [Actionpoint Match Value, AP\\_AMV0](#) register for field information.

### 30.2.18 Actionpoint Match Mask, AP\_AMM4

Address: 0x22D

Access: RW

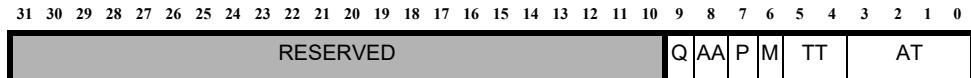
**Figure 30-14 Actionpoint 4 Match Mask Register**

See [Actionpoint Match Mask, AP\\_AMM0](#) register for field information.

### 30.2.19 Actionpoint Control, AP\_AC4

Address: 0x22E

Access: RW

**Figure 30-15 Actionpoint 4 Control Register**

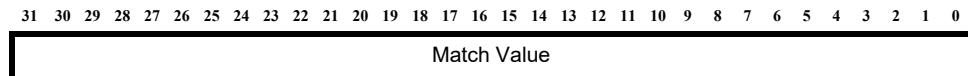
See [Actionpoint Control, AP\\_AC0](#) register for field information.

### 30.2.20 Actionpoint Match Value, AP\_AMV5

Address: 0x22F

Access: RW

**Figure 30-16 Actionpoint 5 Match Value Register**



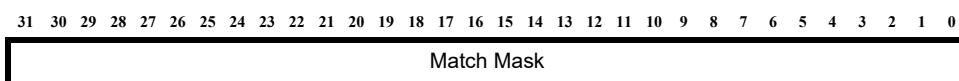
See [Actionpoint Match Value, AP\\_AMV0](#) register for field information.

### 30.2.21 Actionpoint Match Mask, AP\_AMM5

Address: 0x230

Access: RW

**Figure 30-17 Actionpoint 5 Match Mask Register**



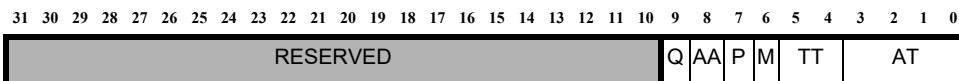
See [Actionpoint Match Mask, AP\\_AMM0](#) register for field information.

### 30.2.22 Actionpoint Control, AP\_AC5

Address: 0x231

Access: RW

**Figure 30-18 Actionpoint 5 Control Register**



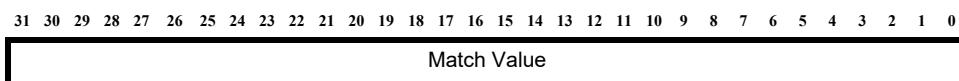
See [Actionpoint Control, AP\\_AC0](#) register for field information.

### 30.2.23 Actionpoint Match Value, AP\_AMV6

Address: 0x232

Access: RW

**Figure 30-19 Actionpoint 6 Match Value Register**

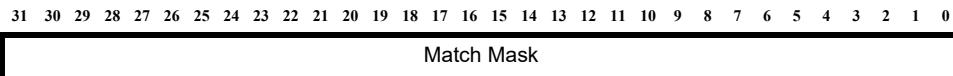


See [Actionpoint Match Value, AP\\_AMV0](#) register for field information.

### 30.2.24 Actionpoint Match Mask, AP\_AMM6

Address: 0x233

Access: RW

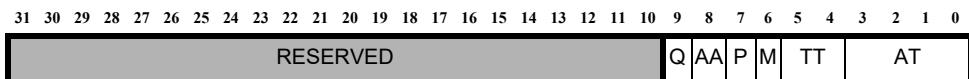
**Figure 30-20 Actionpoint 6 Match Mask Register**

See [Actionpoint Match Mask, AP\\_AMM0](#) register for field information.

### 30.2.25 Actionpoint Control, AP\_AC6

Address: 0x234

Access: RW

**Figure 30-21 Actionpoint 6 Control Register**

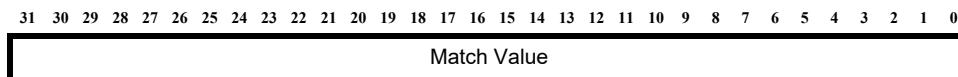
See [Actionpoint Control, AP\\_AC0](#) register for field information.

### 30.2.26 Actionpoint Match Value, AP\_AMV7

Address: 0x235

Access: RW

**Figure 30-22 Actionpoint 7 Match Value Register**



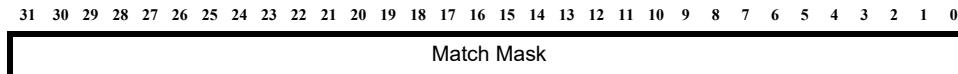
See [Actionpoint Match Value, AP\\_AMV0](#) register for field information.

### 30.2.27 Actionpoint Match Mask, AP\_AMM7

Address: 0x236

Access: RW

**Figure 30-23 Actionpoint 7 Match Mask Register**



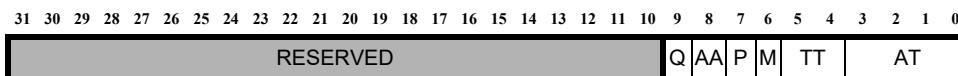
See [Actionpoint Match Mask, AP\\_AMM0](#) register for field information.

### 30.2.28 Actionpoint Control, AP\_AC7

Address: 0x237

Access: RW

**Figure 30-24 Actionpoint 7 Control Register**



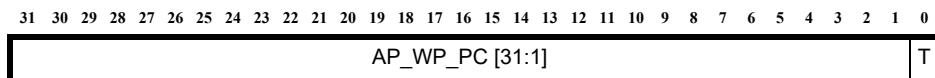
See [Actionpoint Control, AP\\_AC0](#) register for field information.

### 30.2.29 Watchpoint Program Counter, AP\_WP\_PC

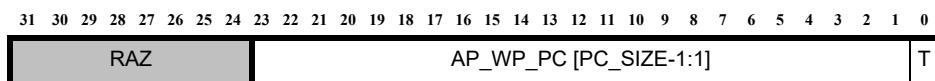
Address: 0x23F

Access: R

**Figure 30-25 AP\_WP\_PC, Watchpoint Program Counter Register when PC\_SIZE = 32**



**Figure 30-26 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.

Actionpoints on external parameters do not have an associated PC and for these actionpoints, the AP\_WP\_PC is undefined. Also, loads from external memory may complete out of order and a watchpoint triggered on such a load does not have the instruction PC loaded in AP\_WP\_PC register.

## 30.3 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.

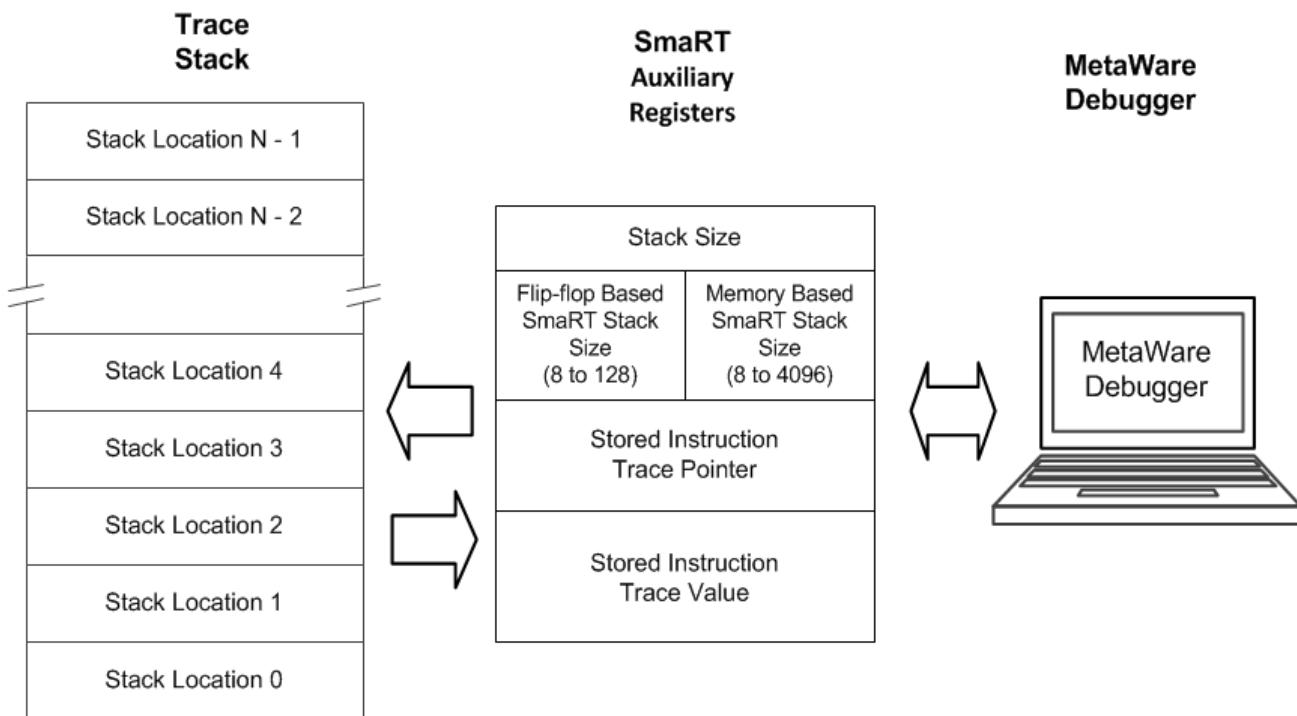
### 30.3.1 Features

- 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.

### 30.3.2 Overview of SmaRT Operation

**Figure 30-27 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

### 30.3.3 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

#### 30.3.4 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 30-28 SMART\_BUILD Value**

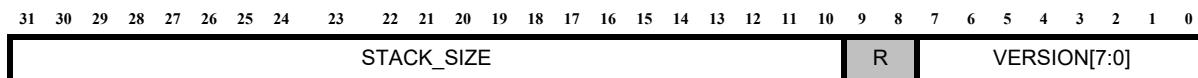


Table 30-10 lists the field descriptions:

**Table 30-10 SMART\_BUILD**

Field	Bits	Description
VERSION	[7:0]	Version of SmaRT component 0x03 = Current version with triplets of registers in each location.
R	[9:8]	Reserved Returns zero when read.

**Table 30-10 SMART\_BUILD**

Field	Bits	Description
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.

### 30.3.5 SmaRT Control Register, SMART\_CONTROL

Address: 0x700

Access: RW

The SMART\_CONTROL register has the following contents:

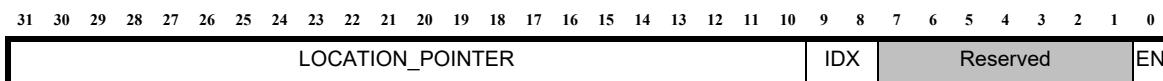
**Figure 30-29 SMART\_CONTROL Register**

Table 30-11 lists the field descriptions:

**Table 30-11 SMART\_CONTROL Register**

Field	Bits	Description
EN	[0]	SmaRT enable Enables the trace mechanism. 0x0 = Trace Disabled. 0x1 = Trace Enabled.
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. 0 = SRC_ADDR value 1 = DEST_ADDR value 2 = FLAGS value 3 = Reserved
LOCATION_POINTER[21:0]	[31:10]	Stack location pointer Sets the value of the stored instruction stack pointer.



Reading an entry outside the specified SmaRT stack size returns 0 data.

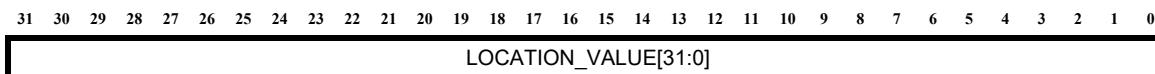
### 30.3.6 SmaRT Data Register, SMART\_DATA

Address: 0x701

Access: R

The SMART\_DATA read-only register has the following contents:

**Figure 30-30 SMART\_DATA Register**



[Table 30-12](#) lists the field descriptions:

**Table 30-12 SMART\_DATA Register**

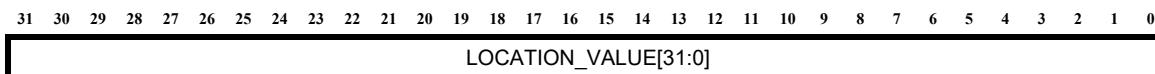
Field	Bits	Description
LOCATION_VALUE	[31:0]	<p>Stored-instruction trace-stack value</p> <p>The value in the stack contained at the location given in the LOCATION_POINTER field and IDX field of the SMART_CONTROL register.</p>

SMART\_DATA register must be read safely only when the processor is in the halt state or in a state in which SmaRT is not collecting data. Reading this register while SmaRT logic is active may return undefined data, because of the intrusive activity of buffer update.

### 30.3.7 SRC\_ADDR Value, Index 0

The SRC\_ADDR value is read from the SMART\_DATA register when the IDX field in the SMART\_CONTROL register is set to 0.

**Figure 30-31 SRC\_ADDR Value**



[Table 30-13](#) lists the field descriptions:

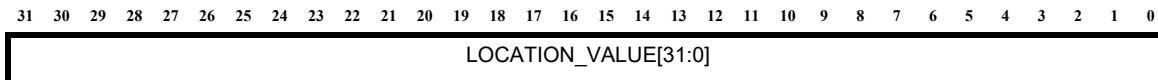
**Table 30-13 SRC\_ADDR Value**

Field	Bits	Description
LOCATION_VALUE	[31:0]	<p>Stored instruction trace stack location value</p> <p>The value of the source address in the stack contained at the location given in the LOCATION_POINTER field of the SMART_CONTROL register.</p>

### 30.3.8 DEST\_ADDR Value, Index 1

The DEST\_ADDR value is read from the SMART\_DATA register when IDX field in the SMART\_CONTROL register is set to 1.

**Figure 30-32 DEST\_ADDR Value Register**



[Table 30-14](#) lists the field descriptions:

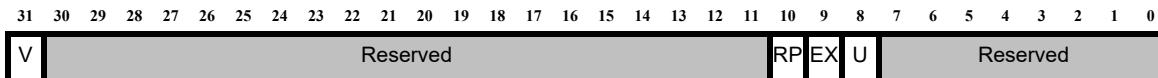
**Table 30-14 DEST\_ADDR Register**

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.

### 30.3.9 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 30-33 FLAGS Value Register**



[Table 30-15](#) lists the field descriptions:

**Table 30-15 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.
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.
Reserved	[30:11]	Reserved Returns zero when read.
V	[31]	Valid Indicates that the value in the stack is valid.

### 30.3.9.1 Reserved Value, Index 3

Zero is read from the SMART\_DATA register when IDX field in the SMART\_CONTROL register is set to 3 .

# 31

## Performance Counters

### 31.1 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 sizeAverage 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.

### 31.1.1 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 1054.

## 31.2 Performance Counters Registers

The following registers provided for interacting with the performance-counter block.

**Table 31-1 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 31-2](#) are provided for interaction with the countable conditions block.

**Table 31-2 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

**Table 31-2 Countable-Conditions Registers**

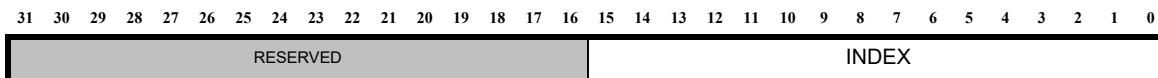
Address	Name	Description
0x242	Countable Conditions Name1 Register, CC_NAME1	Read second four characters of condition name, set by CC_INDEX

### 31.3 Countable Conditions Index Register, CC\_INDEX

Address: 0x240

Access: RW

**Figure 31-1 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 mapping of named conditions to indexes can be discovered using the CC\_INDEX and CC\_NAME0 and CC\_NAME1 registers.

**Table 31-3 CC\_INDEX Field Description**

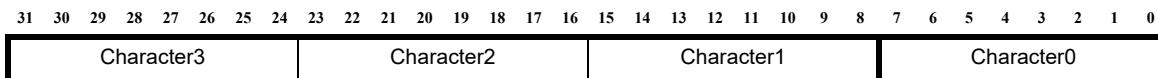
Field	Bit	Description
INDEX	[15:0]	Index of the countable condition whose name is returned through the registers CC_NAME0 and CC_NAME1. The index values start at zero.

## 31.4 Countable Conditions Name0 Register, CC\_NAME0

Address: 0x241

Access: R

**Figure 31-2 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 31-4 CC\_NAME0 Field Description**

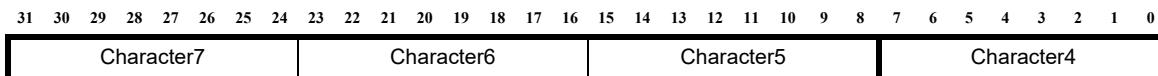
Field	Bit	Description
CC_NAME0	[7:0]	Character 0
CC_NAME0	[15:8]	Character 1
CC_NAME0	[23:16]	Character 2
CC_NAME0	[31:24]	Character 3

## 31.5 Countable Conditions Name1 Register, CC\_NAME1

Address: 0x242

Access: R

**Figure 31-3 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 31-5 CC\_NAME1 Field Description**

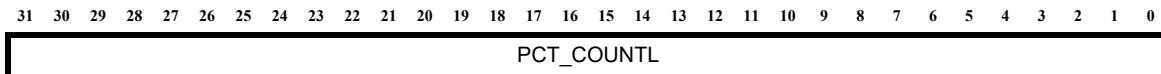
Field	Bit	Description
CC_NAME1	[7:0]	Character 4
CC_NAME1	[15:7]	Character 5
CC_NAME1	[23:16]	Character 6
CC_NAME1	[31:24]	Character 7

## 31.6 Count-Value Registers, PCT\_COUNTL

Address: 0x250

Access: RW

**Figure 31-4 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.

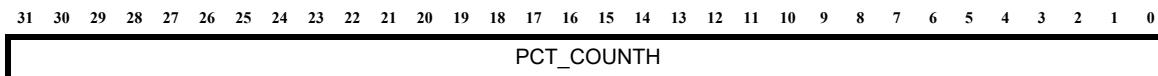
Count values wrap to zero when the maximum count value is reached. If the register is written with a specific value, the counter starts from this value.

## 31.7 Count-Value Registers, PCT\_COUNTH

Address: 0x251

Access: RW

**Figure 31-5 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.

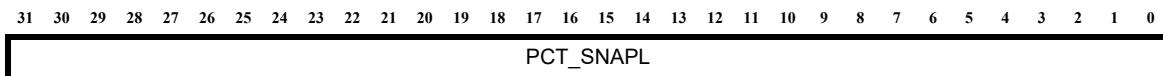
Count values wrap to zero when the maximum count value is reached. If the register is written with a specific value, the counter starts from this value.

## 31.8 Snapshot-Value Registers, PCT\_SNAPL

Address: 0x252

Access: RW

**Figure 31-6 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.

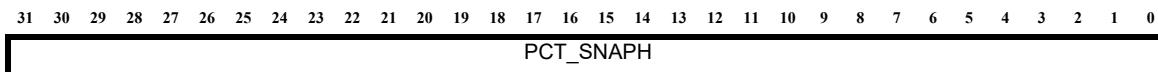
When a counter snapshot is taken, and a counter is in a min or max mode, the contents of the MINMAXL/H registers are copied into the snapshot registers.

## 31.9 Snapshot-Value Registers, PCT\_SNAPH

Address: 0x253

Access: RW

**Figure 31-7 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.

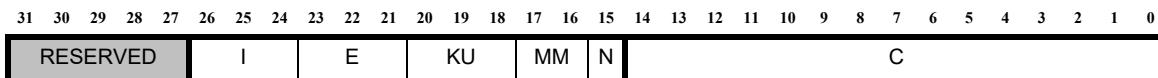
When a counter snapshot is taken, and a counter is in a min or max mode, the contents of the MINMAXL/H registers are copied into the snapshot registers.

### **31.10 Configuration Register, PCT\_CONFIG**

Address: 0x254

Access: RW

**Figure 31-8 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 31-6 PCT\_CONFIG Field Description**

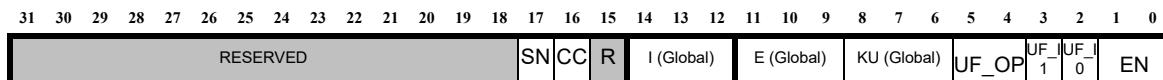
Field	Bit	Description
C	[14:0]	Countable condition index number
N	[15]	Invert condition before use
MM	[17:16]	<p>Defines the counting mode</p> <ul style="list-style-type: none"> <li>■ 0x0: normal counting</li> <li>■ 0x1: min counting</li> <li>■ 0x2: max counting</li> <li>■ 0x3 to 0x4: reserved</li> </ul>
KU	[20:18]	<ul style="list-style-type: none"> <li>■ 0x0: count always</li> <li>■ 0x1: count only when in user mode</li> <li>■ 0x2 : count only when in kernel mode</li> <li>■ 0x3 to 0x7: reserved</li> </ul>
E	[23:21]	<ul style="list-style-type: none"> <li>■ 0x0: count always</li> <li>■ 0x1: count only when an exception is active</li> <li>■ 0x2 : count only when an exception is not active</li> <li>■ 0x3 to 0x7: reserved</li> </ul>
I	[26:24]	<ul style="list-style-type: none"> <li>■ 0x0: count always</li> <li>■ 0x1: count only when an interrupt is taken</li> <li>■ 0x2 : count only when an interrupt is not taken</li> <li>■ 0x3 to 0x7: reserved</li> </ul>

## 31.11 Control Register: PCT\_CONTROL

Address: 0x255

Access: RW

**Figure 31-9 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.

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 31-7 PCT\_CONTROL Field Description**

Field	Bit	Description
EN	[1:0]	<p>Enable condition for all counters</p> <ul style="list-style-type: none"> <li>■ 0x0: Disable counting</li> <li>■ 0x1: Enable counting (global enable)</li> <li>■ 0x2: Count when user flag 0 and user flag 1 conditions (as defined in bits [5:2] of this register) are met.</li> <li>■ 0x3: Global set count run mode (U/K/1, E/NE/1,I/NI/1)</li> </ul> <p>This bit has read and write access.</p>
UF_I0	[2]	<p>0x0: Use user flag 0 as is in the condition UF_OP (bits [5:4])</p> <p>0x1: Invert user flag 0 before using in the condition UF_OP (bits [5:4])</p>
UF_I1	[3]	<p>0x0: Use user flag 1 as is in the condition UF_OP (bits [5:4])</p> <p>0x1: Invert user flag 1 before using in the condition UF_OP (bits [5:4])</p>

**Table 31-7 PCT\_CONTROL Field Description**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
UF_OP	[5:4]	<p>User flag operand</p> <ul style="list-style-type: none"> <li>■ 0x0: enable counting when (<code>User_Flag_0 XOR UF_I0</code>) AND (<code>User_Flag_1 XOR UF_I1</code>) ==1</li> <li>■ 0x1: enable counting when (<code>User_Flag_0 XOR UF_I0</code>) OR (<code>User_Flag_1 XOR UF_I1</code>) ==1</li> <li>■ 0x2: enable counting when (<code>User_Flag_0 XOR UF_I0</code>) ==1</li> <li>■ 0x3: enable counting when (<code>User_Flag_1 XOR UF_I1</code>) ==1</li> </ul>
KU (Global)	[8:6]	<ul style="list-style-type: none"> <li>■ 0x0: count always</li> <li>■ 0x1: count only when in user mode</li> <li>■ 0x2: count only when in kernel mode</li> <li>■ 0x3-0x7: reserved</li> </ul>
E (Global)	[11:9]	<ul style="list-style-type: none"> <li>■ 0x0: count always</li> <li>■ 0x1: count only when an exception is active</li> <li>■ 0x2: count only when an exception is NOT active</li> <li>■ 0x3-0x7: reserved</li> </ul>
I (Global)	[14:12]	<ul style="list-style-type: none"> <li>■ 0x0: count always</li> <li>■ 0x1: count only when an interrupt is active</li> <li>■ 0x2: count only when an interrupt is NOT active</li> <li>■ 0x3-0x7: reserved</li> </ul>
CC	[16]	<p>Clear counts. When this bit is written with 1, all counters are cleared to zero.</p> <p>Write-only – returns zero on read</p>
SN	[17]	<p>Snapshot. When this bit is written with 1, all count values are copied into the snapshot registers.</p> <p>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.</p> <p>Write-only – returns zero on read</p>

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](#)”.

## 31.12 Index-Select Register: PCT\_INDEX

Address: 0x256

Access: RW

**Figure 31-10 PCT\_INDEX Register**



This register is used to select the counters to be accessed through count, snapshot, and configuration registers.

**Table 31-8 PCT\_INDEX Field Description**

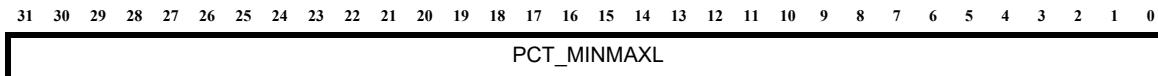
Field	Bit	Description
B[15:0]	[15:0]	Counter number. Selects the counters to be accessed through count, snapshot, and configuration registers. Counter operation is not affected by index selection.

## 31.13 Minimum Value Registers, PCT\_MINMAXL

Address: 0x257

Access: RW

**Figure 31-11 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).

## 31.14 Maximum Value Registers, PCT\_MINMAXH

Address: 0x258

Access: RW

**Figure 31-12 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:

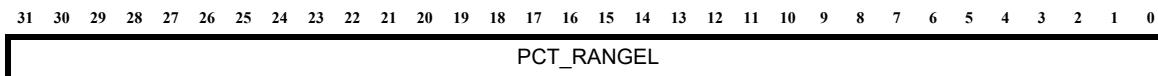
- A sequence is ignored if the condition is true when counters are enabled (by the global-enable register, the global-enable condition, or entry into an address range).
- A sequence is ignored if the condition is true when counters are disabled (by the global-enable register, the global-enable condition, or exit from an address range).

## 31.15 Address-Range Registers, PCT\_RANGEL

Address: 0x259

Access: RW

**Figure 31-13 PCT\_RANGEL 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_RANGEL <= PC`) and either (`PC < PCT_RANGEH`) or (`PCT_RANGEH == 0`).

This register defaults to 0x0 on reset.

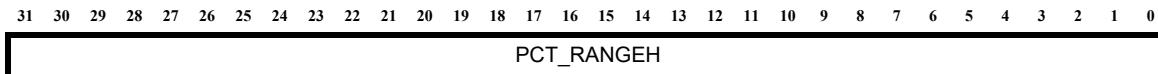
Note that the use of address ranges has effects on min/max behavior. See "[Minimum Value Registers, PCT\\_MINMAXL](#)" and "[Maximum Value Registers, PCT\\_MINMAXH](#)".

## 31.16 Address-Range Registers, PCT\_RANGEH

Address: 0x25A

Access: RW

**Figure 31-14 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_RANGEL <= 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 `0xffffffe+2=0x0`).

This register defaults to 0x0 on reset.

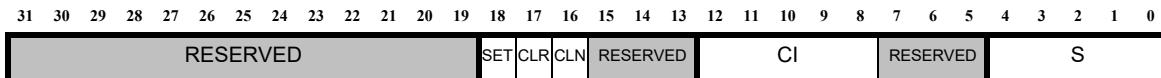
Note that the use of address ranges has effects on min/max behavior. See "[Minimum Value Registers, PCT\\_MINMAXL](#)" and "[Maximum Value Registers, PCT\\_MINMAXH](#)".

## 31.17 User-Flag Register, PCT\_UFLAGS

Address: 0x25B

Access: RW

**Figure 31-15 PCT\_UFLAGS Register**



User-settable flags are provided for instrumenting a specific program. [Table 31-9](#) shows the bit positions for writes.

When the register is read, the current state of the 32-bit flag vector is returned.

The register allows a single aux write to perform both set and clear operations simultaneously.

[Example 31-1](#) describes a few example.

### Example 31-1 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 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, etc.
  - Counters are programmed to count cycles when uflag0/1/2 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, run-time 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/uflag2/... time periods. Different sections can be selected without changing the code.

**Table 31-9 PCT\_UFLAGS Field Description**

Field	Bit	Description
S	[4:0]	Set-flag index
CI	[12:8]	Clear-flag index
CLN	[16]	Clear all flags
CLR	[17]	Clear flag specified by CI field
SET	[18]	Set flag specified by SI field, after clearing operations have taken place

### **31.18 Performance Counter Build-Configuration Register, PCT\_BUILD**

Address: 0xF5

Access: R

**Figure 31-16 PCT\_BUILD Register**



The performance-counter build-configuration register contains the fields shown in [Table 31-10](#) on page [1101](#).

**Table 31-10 PCT\_BUILD Field Description**

Field	Bit	Description
VERSION	[7:0]	<p>Version number.</p> <ul style="list-style-type: none"> <li>■ 0x0: not present</li> <li>■ 0x4: build configuration for ARC EM</li> </ul>
S	[9:8]	<p>Counter size:</p> <ul style="list-style-type: none"> <li>■ 0x0: 32-bit</li> <li>■ 0x1: 48-bit (default)</li> <li>■ 0x2: 64-bit</li> </ul> <p>All other values – reserved</p> <p>Note: The current implementation supports only 48-bit counters.</p>
C	[23:16]	Number of performance counters; 0, 2, 4, 8, and 32 counters are supported. Other values are reserved.
M	[31:24]	Number of counters with min/max capability (starting at counter 0)

## 31.19 Countable Conditions Build Configuration Register, CC\_BUILD

Address: 0xF6

Access: R

**Figure 31-17 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 31-11 CC\_BUILD Field Description**

Field	Bit	Description
VERSION	[7:0]	Version number <ul style="list-style-type: none"> <li>■ 0x0 if no countable conditions are present</li> <li>■ 0x4 for countable conditions for ARC EM</li> </ul>
CC	[31:16]	The number of countable conditions present in the system

# 32

## ARC RTT

### 32.1 ARC RTT Introduction

ARC RTT is a real-time state tracing system for ARCv2-based processors. ARC RTT 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 RTT and display the debug trace.

### 32.2 ARC RTT Auxiliary Registers

**Table 32-1 ARC RTT Registers**

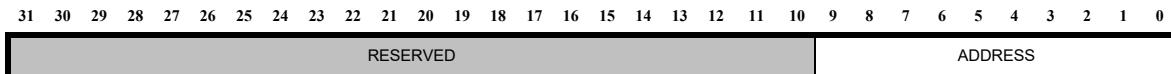
Address	Auxiliary Register Name	Description
0xF2	<a href="#">ARC RTT Build Configuration Register, RTT_BUILD</a>	Build configuration register
0x380	<a href="#">ARC RTT Address Register, RTT_ADDRESS</a>	Contains the ARC RTT memory address that you want to access
0x381	<a href="#">ARC RTT DATA Register, RTT_DATA</a>	Contains the data that you want to read or write to ARC RTT
0x382	<a href="#">ARC RTT CMD Register, RTT_COMMAND</a>	ARC RTT command register

### 32.2.1 ARC RTT Address Register, RTT\_ADDRESS

Address: 0x380

Access: RW

**Figure 32-1 RTT\_ADDRESS Register**



This register specifies the ARC RTT memory address which you want to write to or read from. The address written to this register is presented on the ARC RTT interface address bus; the desired address must be set up prior to writing the RTT\_COMMAND register, which initiates the ARC RTT read or write operation.

The operation that the processor performs on this ARC RTT memory address is specified by the RTT\_COMMAND register.

The RTT\_DATA register contains the data read from this ARC RTT memory address or the data you want to write to this memory address.

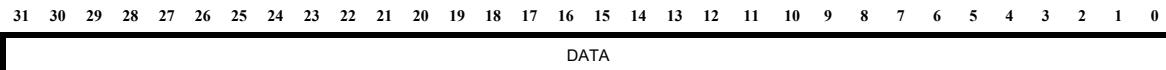
For more information about this register, see the *ARC RTT Databook*.

### 32.2.2 ARC RTT DATA Register, RTT\_DATA

Address: 0x381

Access: RW

**Figure 32-2 RTT\_DATA Register**



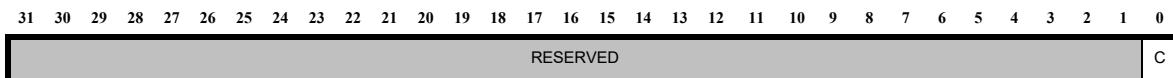
This register contains the data read from the ARC RTT or the data to be written to the ARC RTT. The read or write operation is determined by the value of the RTT\_COMMAND register. The RTT\_ADDRESS register specifies the ARC RTT memory address that is being accessed. For more information about this register, see the *ARC RTT Databook*.

### 32.2.3 ARC RTT CMD Register, RTT\_COMMAND

Address: 0x382

Access: W

**Figure 32-3 RTT\_COMMAND Register**



This register specifies the actions that the processor performs on ARC RTT.

**Table 32-2 RTT\_COMMAND Bit Field Description**

Field	Bit	Description
C	[0]	<b>Specifies the action performed on ARC RTT</b> <ul style="list-style-type: none"><li>■ 0: indicates a read operation from ARC RTT</li><li>■ 1: indicates a write operation to ARC RTT</li></ul>

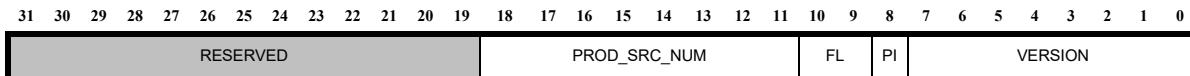
For more information about this register, see the *ARC RTT Databook*.

### 32.2.4 ARC RTT Build Configuration Register, RTT\_BUILD

Address: 0xF2

Access: R

**Figure 32-4 RTT\_BUILD Register**



This register specifies the build-time configuration of ARC RTT.

**Table 32-3 RTT\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of ARC RTT</b> <ul style="list-style-type: none"> <li>■ 0x01: initial version of ARC RTT</li> <li>■ 0x02: second version of ARC RTT</li> <li>■ 0x03: current version</li> </ul>
PI	[8]	<b>Indicates the presence of ARC RTT programming interface</b> This bit is always set to 1; it means that you can program the ARC RTT control registers using this core.
FL	[10:9]	<b>Indicates the feature level of the producer</b> <ul style="list-style-type: none"> <li>■ 00: small producer; defines program counter as a source for trace messages</li> <li>■ 01: medium producer; defines program counter and memory read/write as sources for trace messages</li> <li>■ 10: full producer; defines program counter, core register writes, memory read/write, and auxiliary register read/write as sources for trace messages</li> <li>■ 11: reserved</li> </ul>
PROD_SRC_NUM	[18:11]	<b>Indicates the number assigned to this core by ARC RTT in a multi-core configuration</b> Valid values: 0, 1, 2, 4. Other values are reserved

For more information about this register, see the *ARC RTT Databook*.



# 33

## Power Domain Management and DVFS

### 33.1 Power Domain Management and DVFS Introduction

To reduce static (leakage) power, the ARCV2 ISA provides the capability to divide an ARCV2-based processor into various power domains. The ARCV2 ISA also lets an external DVFS controller unit to dynamically adjust the supply voltage and clock frequency (DVFS) of an ARCV2 processor.

### 33.2 PDM and DVFS Register Interface

**Table 33-1 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 RTT 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	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

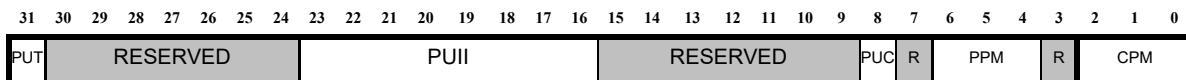
### 33.3 Core Power Status Register, PDM\_PSTAT

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: 0x0

**Figure 33-1 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 33-2 PDM\_PSTAT Field Description**

Field	Bit	Description
CPM	[2:0]	<p>Current power mode of the core. Read-only.</p> <ul style="list-style-type: none"> <li>■ 0x0: Power on mode</li> <li>■ 0x1: PM1 mode</li> <li>■ 0x2: PM2 mode</li> </ul> <p>For more information about the power-down modes, see <a href="#">Table 33-5</a> on page <a href="#">1114</a>.</p>
PPM	[6:4]	<p>Previous power mode of the core. Read-only.</p> <ul style="list-style-type: none"> <li>■ 0x0: Power on mode</li> <li>■ 0x1: PM1 mode</li> <li>■ 0x2: PM2 mode</li> </ul> <p>For more information about the power-down modes, see <a href="#">Table 33-5</a> on page <a href="#">1114</a>.</p>
PUC	[8]	<p>Power up cause. Read-only. This field is relevant only if the core was previously in a power-down mode.</p> <ul style="list-style-type: none"> <li>■ 0x0: Power up by interrupt; for more information about the interrupts that can power up, see the <i>DesignWare ARCv2-based processor Databook</i>.</li> <li>■ 0x1: Power up by debugger</li> </ul>

**Table 33-2 PDM\_PSTAT Field Description**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
PUII	[23:16]	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]	Power-up trigger bit. This bit is write-only and is always read as zero. <ul style="list-style-type: none"> <li>■ Write 1: <ul style="list-style-type: none"> <li>- 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.</li> <li>- When core is in power on state: halt the core.</li> </ul> </li> <li>■ Write 0: No effect; ignored.</li> </ul>

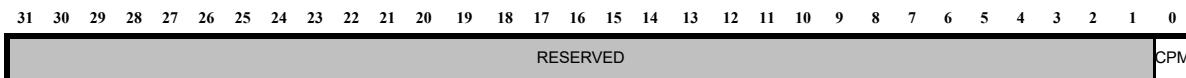
### 33.4 ARC RTT Power Status Register, RTT\_PDM\_PSTAT

Address: 0x611

Access: RW

Reset: 0x0

**Figure 33-2 RTT\_PDM\_PSTAT Register**



Use this register to power down or power up ARC RTT. Reading this register returns the current ARC RTT power status.

**Table 33-3 RTT\_PDM\_PSTAT Field Description**

Field	Bit	Description
CPM	[0]	<p>Write to this bit to power down or power up ARC RTT</p> <ul style="list-style-type: none"> <li>■ 0x0: Send ARC RTT power up request to the external PMU.</li> <li>■ 0x1: Send ARC RTT power down ARC RTT to the external PMU in PM1 mode.</li> </ul> <p>Read this bit to determine the power status of ARC RTT:</p> <ul style="list-style-type: none"> <li>■ 0x0: The external PMU has powered up ARC RTT.</li> <li>■ 0x1: The external PMU has powered down ARC RTT.</li> </ul>

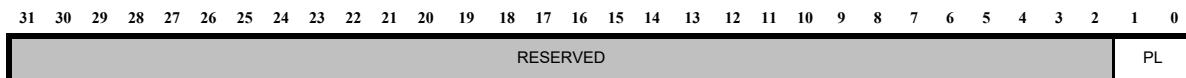
### 33.5 Core Performance Register, DVFS\_PL

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: 0x0

**Figure 33-3 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 33-4 DVFS\_PL Field Description**

Field	Bit	Description
PL	[1:0]	<p>Current performance level of a core.</p> <ul style="list-style-type: none"> <li>■ 0x0: PL0 (performance level 0). This is the default value.</li> <li>■ 0x1: PL1 (performance level 1)</li> <li>■ 0x2: PL2 (performance level 2)</li> <li>■ 0x3: PL3 (performance level 3)</li> </ul> <p>For more information about the performance levels, see the <i>DesignWare ARC EM Databook</i>.</p>

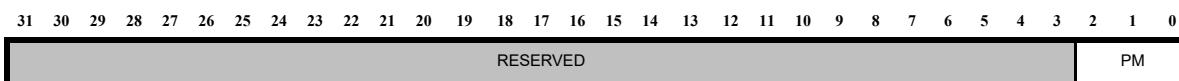
## 33.6 Power Down Register, PDM\_PMODE

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: 0x0

**Figure 33-4 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 33-5 PDM\_PMODE Field Description**

Field	Bit	Description
PM	[2:0]	Power-down mode <ul style="list-style-type: none"> <li>■ 0x0: power on; default</li> <li>■ 0x1: power-down mode 1 (PM1)</li> <li>■ 0x2: power-down mode 2 (PM2)</li> <li>■ 0x3: power-down mode 3 (PM3)</li> </ul>

## 33.7 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.

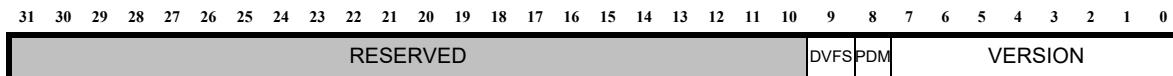
Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCv2-based system.

### 33.8 Power Domain Management and DVFS Build Configuration Register, PDM\_DVFS\_BUILD

Address: 0xF7

Access: R

**Figure 33-5 PDM\_DVFS\_BUILD Register**



This register indicates the presence of the power-domain management and DVFS capabilities in an ARCv2-based processor.

**Table 33-6 PDM\_DVFS\_BUILD Field Description**

Field	Bit	Description
VERSION	[7:0]	Version number 0x2: current version
PDM	[8]	Indicates whether the processor power consumption can be controlled by dividing the processor into different power domains. <ul style="list-style-type: none"> <li>■ 1: The processor can be divided into different power domains.</li> <li>■ 0: The processor is not divided into different power domains.</li> </ul>
DVFS	[9]	Indicates whether the DVFS (dynamic voltage and frequency scaling) functionality is supported. <ul style="list-style-type: none"> <li>■ 1: DVFS is supported.</li> <li>■ 0: DVFS is not supported.</li> </ul>



# 34

## Unit

The floating-point unit consists of a set of hardware accelerated floating-point instructions compliant with IEEE 754-2008 specification.

## 34.1 Core Register Set

Figure 34-1 shows a summary of the core register set.

**Figure 34-1 Core Register Map Summary**

**Figure 34-1 Core Register Map Summary (Continued)**

r62	Long Immediate Data Indicator	
r63	Program Counter [PC_SIZE-1:2], Read-Only (PCL) 0 0	

## 34.2 Extension Core Registers

The register set is extendible in register positions 32-57 (r32-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 34-2 ACCL for fusedMultiplyAdd and fusedMultiplySubtract Operations**

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	Exponent		Significand																												



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 writeback. In this case the writeback value is discarded.

## 34.3 FPU Auxiliary Registers

**Table 34-1 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.

## 34.4 Floating-Point Unit Control Register, FPU\_CTRL

Address: 0x300

Access: RW

Default 0x00000100

**Figure 34-3 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 34-2 FPU\_CTRL Field Description**

Field	Bits	Description
IVE	0	<p>Enable Invalid Operation Exception</p> <ul style="list-style-type: none"> <li>■ 0 : Exception is not raised when an invalid operation occurs and the IV flag in the FPU_STATUS register is set.</li> <li>■ 1 : Exception is raised when an invalid operation occurs and the IV flag in the FPU_STATUS register is not set.</li> </ul>
DZE	1	<p>Enable Divide by Zero Exception</p> <ul style="list-style-type: none"> <li>■ 0 : Exception is not raised when a divide by zero operation occurs and the DZ flag in the FPU_STATUS register is set.</li> <li>■ 1 : Exception is raised when a divide by zero operation occurs and the DZ flag in the FPU_STATUS register is not set.</li> </ul>
RM	[9:8]	<p>Indicates Rounding Mode</p> <ul style="list-style-type: none"> <li>■ 0x0 : Round towards zero; the precise result is rounded to the nearest representable number closest to zero.</li> <li>■ 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.</li> <li>■ 0x2 : Round to positive infinity; the result is rounded to the next most positive representable number.</li> <li>■ 0x3 : Round to negative infinity; the result is rounded to the next most negative representable number.</li> </ul>

## 34.5 Floating-Point Unit Status Register, FPU\_STATUS

Address: 0x301

Access: RW

Reset 0x00000000

**Figure 34-4 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 34-3 FPU\_STATUS Field Description**

Field	Bits	Description
IV	0	<p>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.</p> <ul style="list-style-type: none"> <li>■ If <math>FWE == 0</math>, This bit is read only. Write 1 to the CIV bit to clear this flag.</li> <li>■ If <math>FWE == 1</math>, this bit has both read and write access.</li> </ul>
DZ	1	<p>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.</p> <ul style="list-style-type: none"> <li>■ If <math>FWE == 0</math>, This bit is read only. Write 1 to the CDZ bit to clear this flag.</li> <li>■ If <math>FWE == 1</math>, this bit has both read and write access.</li> </ul>
OF	2	<p>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.</p> <ul style="list-style-type: none"> <li>■ If <math>FWE == 0</math>, This bit is read only. Write 1 to the COF bit to clear this flag.</li> <li>■ If <math>FWE == 1</math>, this bit has both read and write access.</li> </ul>
UF	3	<p>Underflow 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 CUF bit to clear this flag.</p> <ul style="list-style-type: none"> <li>■ If <math>FWE == 0</math>, This bit is read only. Write 1 to the CUF bit to clear this flag.</li> <li>■ If <math>FWE == 1</math>, this bit has both read and write access.</li> </ul>

**Table 34-3 FPU\_STATUS Field Description**

<b>Field</b>	<b>Bits</b>	<b>Description</b>
IX	4	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. <ul style="list-style-type: none"> <li>■ If FWE==0, This bit is read only. Write 1 to the CIX bit to clear this flag.</li> <li>■ If FWE==1, this bit has both read and write access.</li> </ul>
CAL	7	Clear all flags. <ul style="list-style-type: none"> <li>■ If FWE==0, you can write to this bit. Write 1 to clear bits [4;0].</li> <li>■ If FWE==1, this bit read as zero and ignored on write.</li> </ul>
CIV	8	Clear invalid operation flag. <ul style="list-style-type: none"> <li>■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the IV bit.</li> <li>■ If FWE==1, this bit read as zero and ignored on write.</li> </ul>
CDZ	9	Clear divide by zero. <ul style="list-style-type: none"> <li>■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the DZ bit.</li> <li>■ If FWE==1, this bit read as zero and ignored on write.</li> </ul>
COF	10	Clear overflow flag. <ul style="list-style-type: none"> <li>■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the OF bit.</li> <li>■ If FWE==1, this bit read as zero and ignored on write.</li> </ul>
CUF	11	Clear underflow flag. <ul style="list-style-type: none"> <li>■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the UF bit.</li> <li>■ If FWE==1, this bit read as zero and ignored on write.</li> </ul>
CIX	12	Clear inexact flag. <ul style="list-style-type: none"> <li>■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the IX bit.</li> <li>■ If FWE==1, this bit read as zero and ignored on write.</li> </ul>
FWE	31	Flag Write Enable <ul style="list-style-type: none"> <li>■ 0 : 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.</li> <li>■ 1 : 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.</li> </ul>

## 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 signalling 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.

## UF – Underflow

Underflow occurs when the result is too small to be represented by the instruction data format.

## IX – Inexact

Inexact operation occurs when the result of a floating-point operation is rounded and not exact.

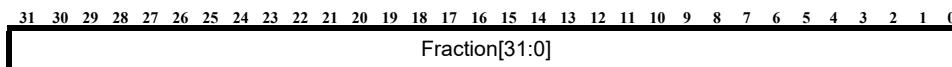
## 34.6 Double-precision Floating Point D1 Lower Register, AUX\_DPFP1L

Address: 0x302

Access: RW

Reset 0x00000000

**Figure 34-5 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 34-4 AUX\_DPFP1L Field Description**

Field	Bit	Description
Fraction	[31:0]	Fractional part of the floating point number

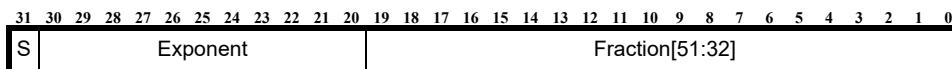
## 34.7 Double-precision Floating Point D1 Higher Register, AUX\_DPFP1H

Address: 0x303

Access: RW

Reset 0x00000000

**Figure 34-6 AUX\_DPFP1H, Double-precision Floating Point D1 Register, Higher 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\_DPFP1H register represents the higher half, bits [63:32], of D1 “[Double-Precision Floating-Point Data Formats](#)”.

**Table 34-5 AUX\_DPFP1H 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 <ul style="list-style-type: none"> <li>■ 0 : Positive number</li> <li>■ 1 : Negative number</li> </ul>

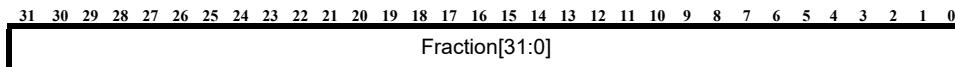
## 34.8 Double-precision Floating Point D2 Lower Register, AUX\_DPFP2L

Address: 0x304

Access: RW

Reset 0x00000000

**Figure 34-7 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 34-6 AUX\_DPFP2L Field Description**

Field	bit	Description
Fraction	[31:0]	Fractional part of the floating point number

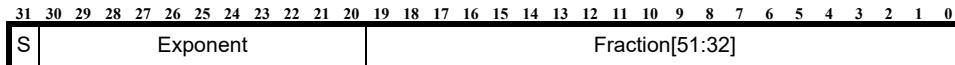
## 34.9 Double-precision Floating Point D2 Higher Register, AUX\_DPFP2H

Address: 0x305

Access: RW

Reset 0x00000000

**Figure 34-8 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 34-7 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 <ul style="list-style-type: none"> <li>■ 0 : Positive number</li> <li>■ 1 : Negative number</li> </ul>

## 34.10 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.

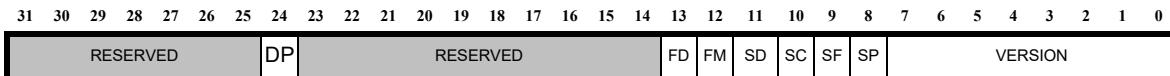
Auxiliary registers, in the range 0x60 to 0x7F and 0xC0 to 0xFF, are assumed to be BCRs. In kernel mode, any read from a non-existent build configuration register in this range returns 0, and no exception is generated. This design enables the kernel-mode code to detect the presence or absence of a BCR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

## 34.11 Floating-Point Unit Build Register, FPU\_BUILD

Address: 0xC8

Access: R

**Figure 34-9 FPU\_BUILD Register**



This register is present only if a floating-point unit is included in the processor configuration. [Table 34-8](#) lists the field descriptions:

**Table 34-8 FPU\_BUILD Field Descriptions**

Field	Bits	Description
VERSION	[7:0]	Floating-Point Unit Version 0x02: Current version. With this version, support for single-precision is always true.
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. <ul style="list-style-type: none"> <li>■ 0: 2 cycle implementation</li> <li>■ 1: 1 cycle implementation</li> </ul>
FD	13	Indicates support for fast divide and square root instructions. <ul style="list-style-type: none"> <li>■ 0: 17 cycle implementation of the divide and square root instructions</li> <li>■ 1: 1 cycle implementation of the divide and square root instructions</li> </ul>
DP	24	Indicates support for double-precision floating-point instructions

# 35

## 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 datatype 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.

## 35.1 XY Data Organization

In an ARC processor with XY memories, the data resides in 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 0x9FFFFFF 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 a 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.

### 35.1.1 Address Generation

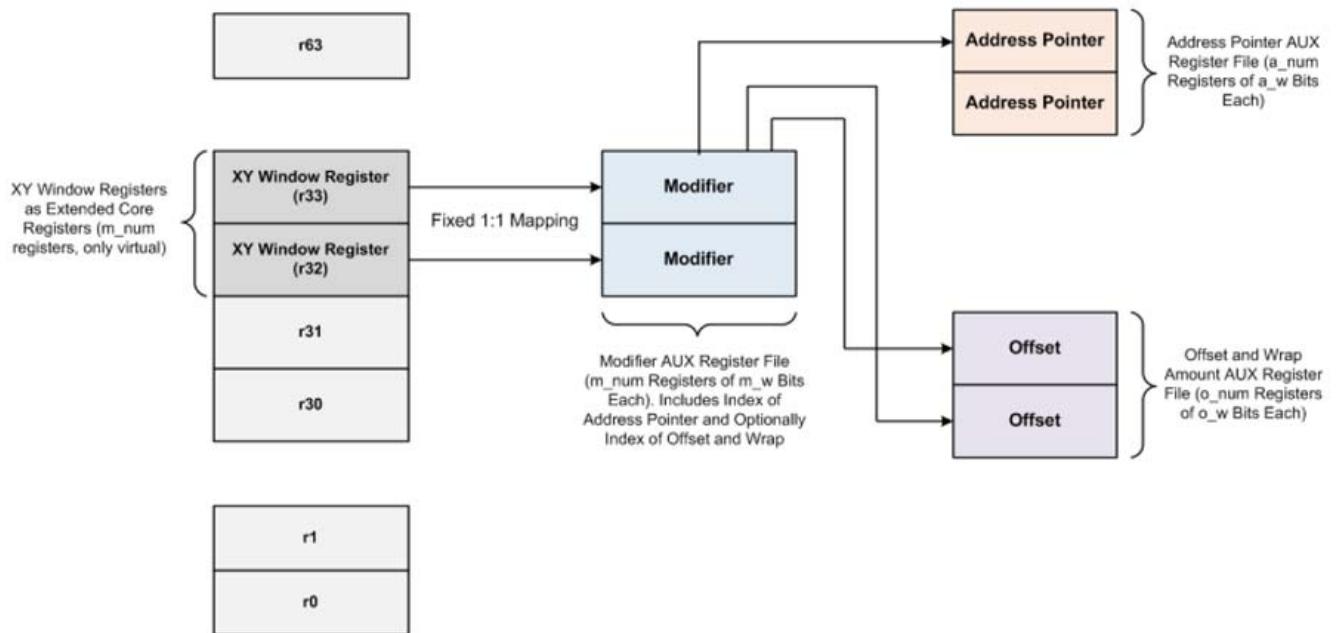
The ARC EM processor provides an implicit memory access to the XY memory through the address generation unit (AGU) as well as explicit access through load and store operations. If you have included an AGU in your processor, and an instruction tries to access the XY memory or DCCM, the processor uses 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 35-1](#) illustrates the AGU register organization.

**Figure 35-1 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.

### 35.1.2 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 writeback 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”

## 35.2 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.



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 35-2 Extended Core Registers when HAS\_AGU==true**

### 35.3 XY Register Set

Table 35-1 XY Registers

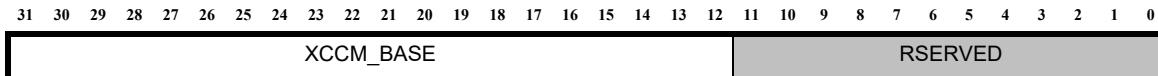
Address	Name	Description
0x5F8	XCCM Base Address, XCCM_BASE	XCCM base address
0x5F9	YCCM Base Address, YCCM_BASE	YCCM base address
0x79	XY Build Configuration Register, XY_BUILD	XY Build Configuration Register

### 35.3.1 XCCM Base Address, XCCM\_BASE

Address: 0x05F8

Access: RW

**Figure 35-3 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.

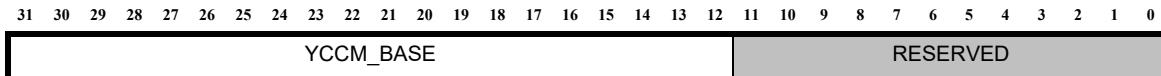
- If the address bus width is 16, the fields [15:12] indicate the XCCM memory region.
- If the address bus width is 20, the fields [19:16] indicate the XCCM memory region.
- If the address bus width is 24, the fields [23:20] indicate the XCCM memory region.
- If the address bus width is 32, the fields [31:28] indicate the XCCM memory region.

### 35.3.2 YCCM Base Address, YCCM\_BASE

Address: 0x5F9

Access: RW

**Figure 35-4 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.

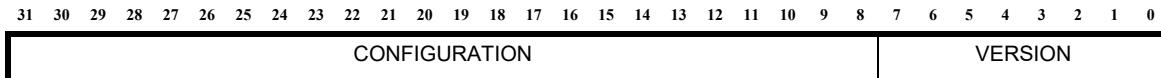
- If the address bus width is 16, the fields [15:12] indicate the YCCM memory region.
- If the address bus width is 20, the fields [19:16] indicate the YCCM memory region.
- If the address bus width is 24, the fields [23:20] indicate the YCCM memory region.
- If the address bus width is 32, the fields [31:28] indicate the YCCM memory region.

### 35.3.3 XY Build Configuration Register, XY\_BUILD

Address: 0x79

Access: R

**Figure 35-5 XY\_BUILD Register**



**Table 35-2 XY\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of the XY unit</b> 0x20: Current version
CONFIGURATION	[9:8]	<b>XY memory configuration</b> <ul style="list-style-type: none"> <li>■ 0: One physical memory: DCCM.</li> <li>■ 1: Two physical memories: one for DCCM+X and one for Y.</li> <li>■ 2: Three physical memories: one each for DCCM, X, and Y.</li> </ul>
	[10]	<b>Indicates if the XY memory supports interleaving</b> <ul style="list-style-type: none"> <li>■ 1: XY memories support interleaving; the X and Y memories are split into even and odd instances to enable single-cycle unaligned accesses.</li> <li>■ 0: XY memories do not support interleaving.</li> </ul>
	[11]	<b>Reserved</b>
	[15:12]	<b>XY memory size</b> <ul style="list-style-type: none"> <li>■ 0x0: 4 KB</li> <li>■ 0x1: 8 KB</li> <li>■ 0x2: 16 KB</li> <li>■ 0x3: 32 KB</li> <li>■ 0x4: 64 KB</li> </ul>
	[31:16]	<b>Reserved</b>



# 36

## 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. The address generation unit (AGU) is used in these implementations. The AGU logic spans the following pipeline stages:

- RF stage: read the value of the address pointer and return as a source operand, while post-incrementing the address pointer
- X1-X3: keep track of the old address pointer value which is restored on a pipeline restart event
- CA: commit the address pointer update

For module buffering, the AGU pipeline forwards two address pointers and a wrap signal to the DMP:

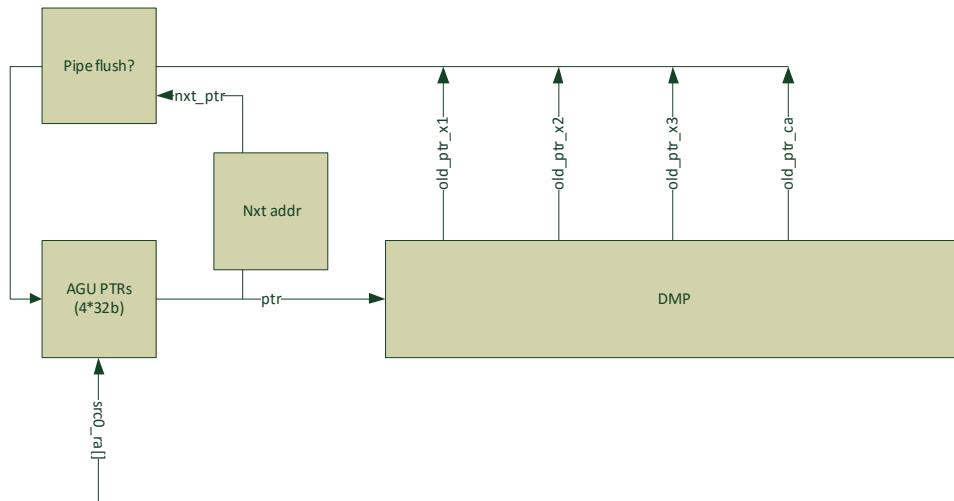
1. Primary address pointer: pointing to the start of the data
2. Secondary address pointer: pointing to the base of the buffer for wrapping data
3. Wrap signal: if true then the secondary part of the data needs to come from the base of the buffer



**Note** The last part of unaligned data will not come from the next linear address but from the secondary address pointer. Modulo buffers need to be aligned at 64b boundaries to ensure that unaligned wrapping data is spread across an even and odd memory bank.

Instructions using an AGU as an operand stall in RF if there is any pending SR operation.

Figure 36-1 illustrates the AGU pipeline. The address pointer register file is implemented in the operand stage. The register file is updated speculatively while forwarding the old pointer value to the DMP. The old pointer value is propagated through the DMP pipe.



**Figure 36-1 Figure 5 AGU Pipeline**

In case of a (partial) pipeline flush even the old address pointer values are used to reset the speculated pointer value. Pipeline flush events can originate from different stages; depending on the stage the AGU pointer need to be fully rolled back and restored or partially restored (X2 vs WA).

## 36.1 AGU Auxiliary Register Set

Table 36-1 AGU Registers

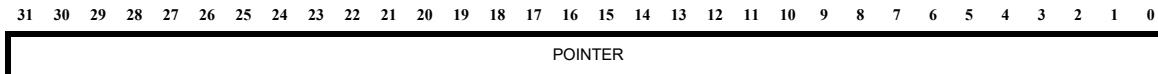
Address	Name	Access	Description
0xCC	AGU Build Configuration Register, AGU_BUILD	R	AGU Build Configuration Register
0x5C0 to 0x5CF	AGU Address Pointer Registers, AGU_AUX_APx	rw	Address generation unit address pointer registers
0x5D0 to 0x5DF	AGU Offset Registers, AGU_AUX_OSx	rw	Address generation unit offset registers
0x5E0 to 0x5F7	AGU Modifier Registers, AGU_AUX_MODx	rw	Address generation unit modifier registers

### 36.1.1 AGU Address Pointer Registers, AGU\_AUX\_APx

Address: 0x5C0 to 0x5CB

Access: rw

**Figure 36-2 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 36-2 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 XY 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 condition is true.

If a condition is false, the instruction is skipped, and the address pointer registers are not updated, and there are no XY memory loads; both the source and destination addresses retain the original values from before the update.

**Table 36-3 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 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 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 AGU_AUX_MODx. [15:14].
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 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 36-3 Post-Incrementing Address Pointer Modes (Continued)**

<b>AGU_AUX_MODx.[31:29) Value</b>	<b>Description</b>
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	<ul style="list-style-type: none"> <li>■ 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:           <pre>xy_ptr = AGU_AUX_AP[ptr_reg] AGU_AUX_AP[sec_ptr_reg] = xy_ptr + AGU_AUX_OS[offset_reg]</pre> </li> <li>■ 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:           <pre>xy_ptr = AGU_AUX_AP[ptr_reg] AGU_AUX_AP[sec_ptr_reg] = xy_ptr + offset_imm</pre> </li> </ul>

 **Note**

- Address pointer value is updated only if the offset is non-zero or if the 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 36-4 AGU\_AUX\_APx Field Description**

<b>Field</b>	<b>Bit</b>	<b>Description</b>
POINTER	[31:0]	Byte address pointers to values in the XY or DCCM memories

### Example 36-1 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 = unmber 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
    // 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;
}
```

### 36.1.2 AGU Offset Registers, AGU\_AUX\_OSx

Address: 0x5D0 to 0x5D7

Access: rw

**Figure 36-3 AGU\_AUX\_OSx Register**



These registers are used to determine the increment values and wrap boundaries during the XY 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 36-5 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 36-3 on page 1143.

### **36.1.3 AGU Modifier Registers, AGU\_AUX\_MODx**

Address: 0x5E0 to 0x5F7

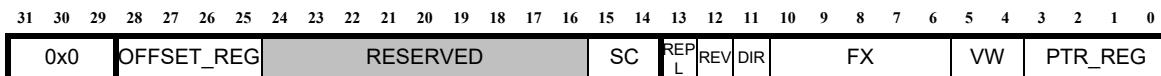
Access: rw

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 register allows 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.

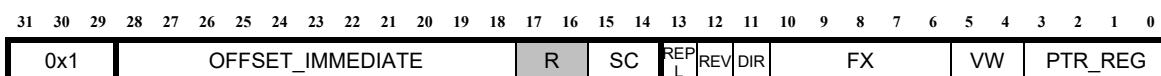
**Table 36-6 Number of Modifier Registers**

<b>AGU Configuration (AGU_SIZE)</b>	<b>Number of Modifier Registers</b>
small	4
medium	12
large	24

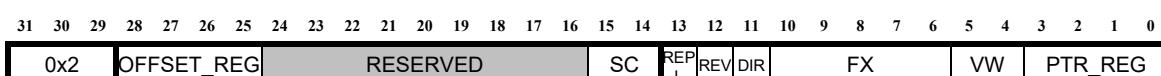
**Figure 36-4 AGU\_AUX\_MODx Register with Offset Register Increment**



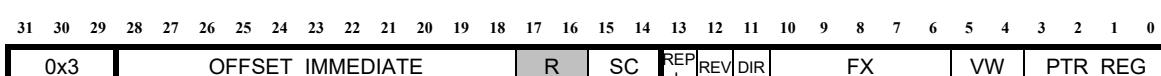
**Figure 36-5 AGU AUX MODx Register with Immediate Offset**



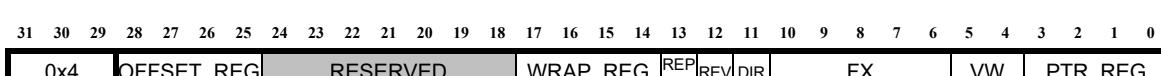
**Figure 36-6 AGU\_AUX MODx Register with Bit Reverse Increment using Offset Register**

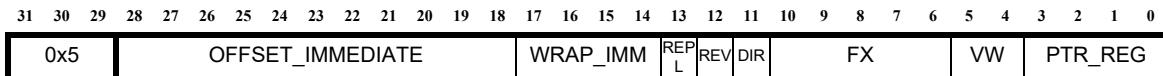
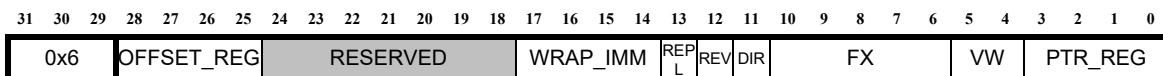
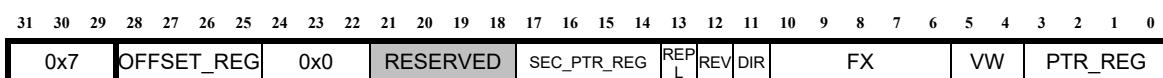
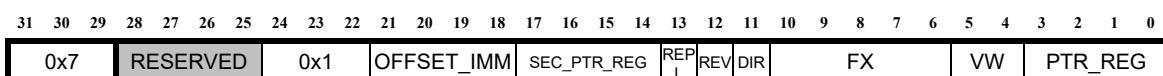


**Figure 36-7 AGU\_AUX\_MODx Register with Bit Reverse Increment using Immediate Offset**



**Figure 36-8** AGU\_AUX\_MDRx Register with Module Wrapping using Offset and Wrap Registers



**Figure 36-9 AGU\_AUX\_MODx Register with Modulo Wrapping using Immediate Offset and Wrap****Figure 36-10 AGU\_AUX\_MODx Register with Modulo Wrapping using Offset Register and Immediate Wrap****Figure 36-11 AGU\_AUX\_MODx Register when Performing Pointer Arithmetic with Offset Register****Figure 36-12 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. <ul style="list-style-type: none"> <li>■ 1: decrement the address pointer</li> <li>■ 0: increment the address pointer</li> </ul>
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: <ul style="list-style-type: none"> <li>■ lower_boundary == base address of the XY memory region</li> <li>■ upper_boundary == lower_boundary + <math>2^{WRAP\_IMM+1}</math></li> </ul> 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 $OPC == 5   6$ .
WRAP_REG	[17:14]	Specifies the index of the wrap register that holds the unsigned wrap value. This field is available only if $OPC == 4$ . $\text{upper\_boundary} == \text{lower\_boundary} + \text{offset}[WRAP\_REG]$
SUB_OP	[24:22]	This field is specific to $OPC == 7$ . Specifies whether an offset register or offset immediate is used in the pointer arithmetic.

Field	Bit	Description
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 $OPC == 1   3   6   7$ . When $OPC = 7$ , the OFFSET_IMM field is at position [21:19], the bits [24:22] are taken up by the SUB_OP 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 $OPC == 0   2   4   5   7$ .

Field	Bit	Description
OPC	[31:29]	<p>This field controls the arithmetic to increment or decrement the address pointer.</p> <ul style="list-style-type: none"> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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.</li> <li>■ 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:</li> </ul> <pre>xy_ptr = AGU_AUX_AP[ptr_reg] AGU_AUX_AP[sec_ptr_reg] = xy_ptr + AGU_AUX_OS[offset_reg]</pre> <ul style="list-style-type: none"> <li>■ 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:</li> </ul> <pre>xy_ptr = AGU_AUX_AP[ptr_reg] AGU_AUX_AP[sec_ptr_reg] = xy_ptr + offset_imm</pre> <p>Note: For a description of the bit-reverse increment and modulo wrapping increment methods, see "<a href="#">Address Generation</a>" on page <a href="#">1130</a>.</p>

## 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 36-7](#) lists the modifier register fields that define the data transformation you can perform:

**Table 36-7 Data Transformation-Related Fields in the Modifier Registers**

Field	Bit	Description
VW	[5:4]	Specifies the vector width in bytes.
FX	[10:6]	Data and container packing information for the source operands; See <a href="#">Table 36-8</a> . 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 36-8 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
11	32	32	The container is the same size as the data

**Table 36-8 Data Transformation Modes Based on the FX Field**

<b>Fx</b>	<b>Container Size in Bits</b>	<b>Data Size in Bits</b>	<b>Expand and Align</b>
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 LSB 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 LSB 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:

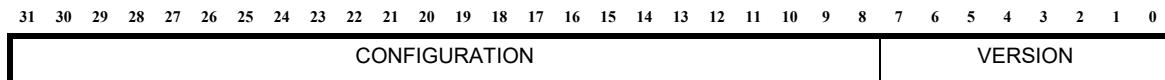
- Limit the width of the writeback operation
- Swap data in containers (for example, endian conversion)
- Reverse vector container order (for example, endian conversion)
- Replicate a container across a vectorYCCM\_BASE

### 36.1.4 AGU Build Configuration Register, AGU\_BUILD

Address: 0xCC

Access: R

**Figure 36-13 AGU\_BUILD Register**



This register indicates the presence of the Address Generator Unit (AGU) and its configuration.

**Table 36-9 AGU\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of the AGU unit</b> 0x01: First version 0x02: Current version; supports copy and arithmetic on the AGU address pointers.
CONFIGURATION	[11:8]	<b>Number of address pointers</b> The number of address pointers is defined by the build option <code>agu_size</code> . <ul style="list-style-type: none"> <li>■ If <code>agu_size==small</code>; number of address pointers = 4</li> <li>■ If <code>agu_size==medium</code>; number of address pointers = 8</li> <li>■ If <code>agu_size==large</code>; number of address pointers = 12</li> </ul>
	[15:12]	<b>Number of offset registers</b> The number of offset registers is defined by the build option <code>agu_size</code> . <ul style="list-style-type: none"> <li>■ If <code>agu_size==small</code>; number of address pointers = 2</li> <li>■ If <code>agu_size==medium</code>; number of address pointers = 4</li> <li>■ If <code>agu_size==large</code>; number of address pointers = 8</li> </ul>
	[20:16]	<b>Number of modifiers</b> The number of modifier registers is defined by the build option <code>agu_size</code> . <ul style="list-style-type: none"> <li>■ If <code>agu_size==small</code>; number of address pointers = 4</li> <li>■ If <code>agu_size==medium</code>; number of address pointers = 12</li> <li>■ If <code>agu_size==large</code>; number of address pointers = 24</li> </ul>
	[23:21]	<b>Write buffer size</b> <ul style="list-style-type: none"> <li>■ 0x2: the buffer can store 2 x 32 bits</li> <li>■ 0x4: the buffer can store 4 x 32 bits</li> </ul>
	[24]	Indicates the presence of the accordion pipeline stage <ul style="list-style-type: none"> <li>■ 1: the accordion pipeline stage is included; the XY-logic is included as a separate stage in the pipeline.</li> <li>■ 0: the accordion pipeline stage is not included; the XY-logic becomes part of the fetch stage in the pipeline.</li> </ul>
	[31:25]	<b>Reserved</b>



# 37

## Bitstream Processing

Use the auxiliary registers in [Table 37-1](#) to program and control bitstreams.

**Table 37-1 Bitstream Auxiliary Registers**

Address	Name	Access	Description
0x5FC	<a href="#">Bitstream Control Register, BS_AUX_CTRL</a>	RW	Bitstream control register.
0x5FD	<a href="#">Bitstream Base Address Register, BS_AUX_ADDR</a>	RW	Bitstream base address register
0x5FE	<a href="#">Bitstream Offset Register, BS_AUX_BIT_OS</a>	RW	Bitstream offset register
0x5FF	<a href="#">Bitstream Write Data Register, BS_AUX_WDATA</a>	RW	Bitstream data register

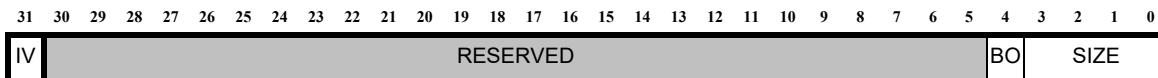
### 37.0.1 Bitstream Control Register, BS\_AUX\_CTRL

Address: 0x5FC

Access: RW

Reset: 0x00

**Figure 37-1 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 37-2 BS\_AUX\_CTRL Field Description**

Field	Bit	Description
SIZE	[3:0]	Specifies the size of the XY memory buffer in KB Buffer size = $2^{\text{SIZE}}$ KB.
BO	4	Specifies the bit ordering in a byte. <ul style="list-style-type: none"> <li>■ 1: Bitstream starts at the MSB.</li> <li>■ 0: Bitstream starts at the LSB.</li> </ul>
IV	31	Invalidate the prefetched data <ul style="list-style-type: none"> <li>■ 1: Prefetched data is invalidated and the bit offset register is set to 0</li> </ul> When read, this bit always reads 0.

**Example 37-1 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

```

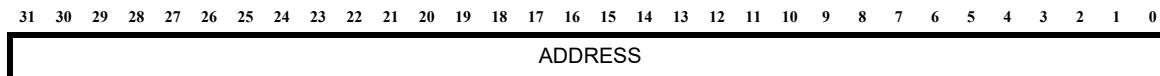
### 37.0.2 Bitstream Base Address Register, BS\_AUX\_ADDR

Address: 0x5FD

Access: RW

Reset: 0x00

**Figure 37-2 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 37-3 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.

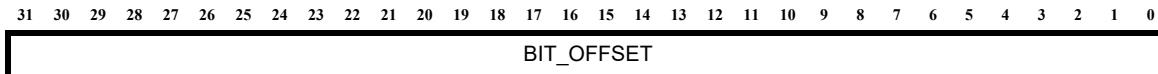
### 37.0.3 Bitstream Offset Register, BS\_AUX\_BIT\_OS

Address: 0x5FE

Access: RW

Reset: 0x00

**Figure 37-3 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 37-4 BS\_AUX\_BIT\_OS Field Description**

Field	Bit	Description
BIT_OFFSET	[31:0]	Specifies the bit offset from the bitstream base address.

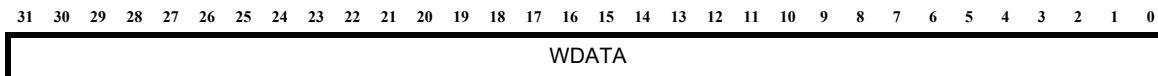
### 37.0.4 Bitstream Write Data Register, BS\_AUX\_WDATA

Address: 0x5FF

Access: RW

Reset: 0x00

**Figure 37-4 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 37-5 BS\_AUX\_WDATA Field Description**

Field	Bit	Description
WDATA	[31:0]	Stores the partial write data for bitstream operations.

## 37.1 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 37-6 Build Configuration Registers**

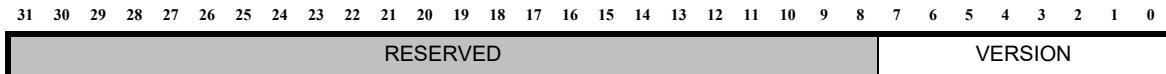
Number	Name	Access	Description
0xCB	Bitstream Identity Register, BS_BUILD	R	Build configuration register: bitstream interface

### 37.1.1 Bitstream Identity Register, BS\_BUILD

Address: 0xCB

Access: R

**Figure 37-5 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 37-7 BS\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of the bitstream interface</b> 0x01: Initial version 0x02: Current version

## 37.2 Exceptions

See [Table 4-7](#) on page [226](#).

# 38

## 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"](#).

### 38.1 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.



# 39

## ARConnect

### 39.1 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.

### 39.2 ARConnect Register Interface

**Table 39-1 ARConnect Registers**

Address	Name	Description
0X600	MCIP_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	MCIP_WDATA	Write data register. Each core uses this register to write data to ARConnect. This register is present only when ARConnect is configured.
0x602	MCIP_READBACK	Read data register. Each core uses this register to read data from ARConnect. This register is present only when ARConnect is configured.
0xD0	MCIP_SYSTEM_BUILD	Build configuration for: ARConnect System
0xD1	MCIP_SEMAPHORE_BUILD	Build configuration for: ARConnect inter-core semaphore unit
0xD2	MCIP_MESSAGE_BUILD	Build configuration for: ARConnect inter-core message unit
0xD3	MCIP_PMU_BUILD	Build configuration for: ARConnect power management unit
0xD5	MCIP_IDU_BUILD	Build configuration for: inter-core interrupt distribution unit

**Table 39-1 (Continued)ARConnect Registers**

Address	Name	Description
0xD6	<a href="#">CONNECT_GFRC_BUILD</a>	Build configuration for: ARConnect global free running counter
0xE0	<a href="#">MCIP_ICI_BUILD</a>	Build configuration for: ARConnect inter-core interrupt unit
0xE1	<a href="#">MCIP_ICD_BUILD</a>	Build configuration for: ARConnect inter-core debug unit
0xE2	<a href="#">MCIP_ASI_BUILD</a>	Build configuration for: ARConnect slave interface unit
0xE3	<a href="#">CONNECT_PDM_BUILD</a>	Build configuration for: ARConnect power domain management unit

### 39.3 ARConnect Command Register, MCIP\_CMD

Address: 0x600

Access: RW

**Figure 39-1 MCIP\_CMD 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					PARAMETER[15:0]		COMMAND[7:0]																								

The MCIP\_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 MCIP\_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 MCIP\_CMD register to check the content of the last written command. In earlier releases, this register was named MCIP\_CMD.

**Table 39-2 MCIP\_CMD Register Field Description**

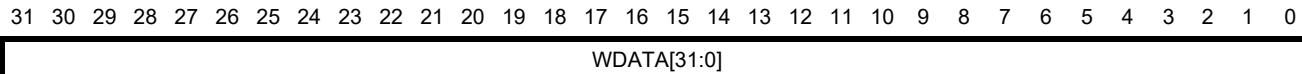
Field	Bits	Description
COMMAND	[7:0]	Command to ARConnect. The list of commands is described in the <i>DesignWare ARCv2-based processor Databook</i> .
PARAMETER	[23:8]	Parameter field of the command to ARConnect. Any parameter that is required by a command is supplied in this field.

## 39.4 ARConnect Write Data Register, MCIP\_WDATA

Address: 0x601

Access: RW

**Figure 39-2 MCIP\_WDATA Register**



The MCIP\_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 MCIP\_WDATA register depends on the ARConnect command being issued. In earlier releases, this register was named MCIP\_WDATA.

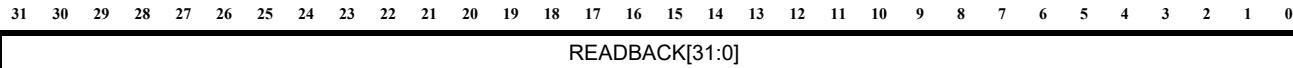
For more information about the layout, see the *DesignWare ARCv2-based Processor Databook*.

## 39.5 ARConnect Read Data Register, MCIP\_READBACK

Address: 0x602

Access: R

**Figure 39-3 MCIP\_READBACK Register**



The MCIP\_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 MCIP\_READBACK register depends on the issued ARConnect command. In earlier releases, this register was named MCIP\_READBACK.

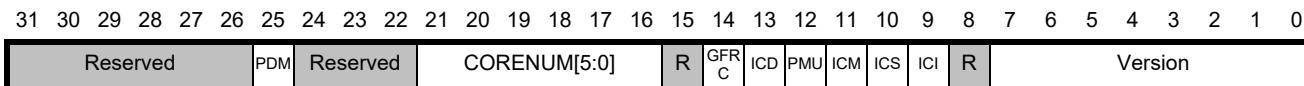
For more information about the layout, see the *DesignWare ARCv2-based Processor Databook*.

### 39.5.1 ARConnect Build Configuration Register, MCIP\_SYSTEM\_BUILD

Address: 0xD0

Access: R

**Figure 39-4 MCIP\_SYSTEM\_BUILD Register**



The MCIP\_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 39-3 MCIP\_SYSTEM\_BUILD Register Field Description**

Field	Bits	Description
Version	[7:0]	The ARConnect version. <ul style="list-style-type: none"> <li>■ 0x1: First version</li> <li>■ 0x2: Current version; <ul style="list-style-type: none"> <li>- Changed the starting CORE_ID to 0 from 1 (first version).</li> <li>- Renamed the component from MCIP to ARCONNECT.</li> </ul> </li> </ul>
ASI	[8]	<ul style="list-style-type: none"> <li>■ 0x0: No ARConnect slave unit</li> <li>■ 0x1: Has ARConnect slave unit</li> </ul>
ICI	[9]	<ul style="list-style-type: none"> <li>■ 0x0: No inter-core interrupt unit</li> <li>■ 0x1: Has inter-core interrupt unit</li> </ul>
ICS	[10]	<ul style="list-style-type: none"> <li>■ 0x0: No inter-core semaphore unit</li> <li>■ 0x1: Has inter-core semaphore unit</li> </ul>
ICM	[11]	<ul style="list-style-type: none"> <li>■ 0x0: No inter-core message unit</li> <li>■ 0x1: Has inter-core message unit</li> </ul>
PMU	[12]	<ul style="list-style-type: none"> <li>■ 0x0: No power management unit</li> <li>■ 0x1: Has power management unit</li> </ul>
ICD	[13]	<ul style="list-style-type: none"> <li>■ 0x0: No inter-core debug unit</li> <li>■ 0x1: Has inter-core debug unit</li> </ul>
GFR_C	[14]	<ul style="list-style-type: none"> <li>■ 0x0: No global free-running counter</li> <li>■ 0x1: Has global free-running counter</li> </ul>

**Table 39-3 MCIP\_SYSTEM\_BUILD Register Field Description**

Field	Bits	Description
CORENUM	[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.
IDU	[23]	<ul style="list-style-type: none"><li>■ 0x0: No interrupt distribution unit</li><li>■ 0x1: Has interrupt distribution unit</li></ul>
PDM	[25]	<ul style="list-style-type: none"><li>■ 0x0: No ARConnect power domain management unit</li><li>■ 0x1: Has ARConnect power domain management unit</li></ul>

### 39.5.2 Inter-core Semaphore Unit Build Configuration Register, MCIP\_SEMA\_BUILD

Address: 0xD1

Access: R

**Figure 39-5 MCIP\_SEMA\_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																																			

The MCIP\_SEMA\_BUILD register is an optional BCR register. If the Inter-Core Semaphore Unit is not configured in ARConnect, reading the MCIP\_SEMA\_BUILD register returns zero. In earlier releases, this register was named MCIP\_SEMA\_BUILD.

**Table 39-4 MCIP\_SEMA\_BUILD Register Field Description**

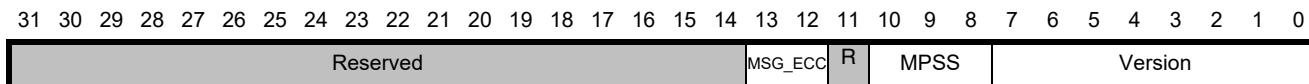
Field	Bits	Description
Version	[7:0]	Inter-core Semaphore Unit version. 0x1: Current version
NUMSEMAS	[9:8]	Number of Semaphores: <ul style="list-style-type: none"> <li>■ 2'b00: 8 semaphores</li> <li>■ 2'b01: 16 semaphores</li> <li>■ 2'b10: 32 semaphores</li> </ul>

### 39.5.3 Inter-core Message Unit Build Configuration Register, MCIP\_MESSAGE\_BUILD

Address: 0xD2

Access: R

**Figure 39-6 MCIP\_MESSAGE\_BUILD Register**



The MCIP\_MESSAGE\_BUILD register is an optional BCR register. If the Inter-Core Message Unit is not configured in ARConnect, reading the MCIP\_MESSAGE\_BUILD register returns zero. In earlier releases, this register was named MCIP\_MESSAGE\_BUILD.

**Table 39-5 MCIP\_MESSAGE\_BUILD Register Field Description**

Field	Bits	Description
Version	[7:0]	Inter-core Message Unit version. 0x01: Current version.
MPSS	[10:8]	Size of the message-passing SRAM <ul style="list-style-type: none"> <li>■ 3'b000: 128 B</li> <li>■ 3'b001: 256 B</li> <li>■ 3'b010: 512 B</li> <li>■ 3'b011: 1 KB</li> <li>■ 3'b100: 2 KB</li> <li>■ 3'b101: 4 KB</li> </ul>
MSG_ECC	[13:12]	ECC or parity protection scheme of Message SRAM <ul style="list-style-type: none"> <li>■ 2'b00: No protection</li> <li>■ 2'b01: Parity protection</li> <li>■ 2'b10: ECC (SECDED) protection</li> </ul>

### 39.5.4 Power Management Unit Build Configuration Register, MCIP\_PMU\_BUILD

Address: 0xD3

Access: R

**Figure 39-7 CONNECT\_PMU\_BUILD Register**



The MCIP\_PMU\_BUILD register is an optional BCR. If the PMU is not configured in ARConnect, reading the MCIP\_PMU\_BUILD register returns zero. In earlier releases, this register was named MCIP\_PMU\_BUILD.

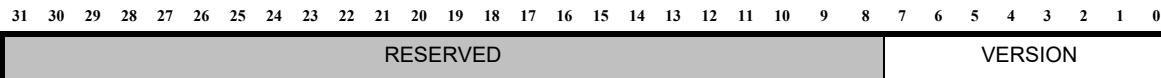
**Table 39-6 CONNECT\_PMU\_BUILD Field Description**

Field	Bit	Description
VERSION	[7:0]	<p>Version number</p> <ul style="list-style-type: none"> <li>■ 0x1: First version</li> <li>■ 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.</li> <li>■ 0x3: Current version; Added two additional counters (RSTCNT and PDCNT) and four relevant commands: CMD_PMU_SET_RSTCNT, CMD_PMU_READ_RSTCNT, CMD_PMU_SET_PDCNT, CMD_PMU_READ_PDCNT</li> </ul>

### 39.5.5 ARConnect Global Free Running Counter Build Configuration Register, MCIP\_GFRC\_BUILD

Address: 0xD6

Access: R

**Figure 39-8 MCIP\_GFRC\_BUILD Register**

The MCIP\_GFRC\_BUILD register is an optional BCR. If the global free running counter is not configured in ARConnect, reading the MCIP\_GFRC\_BUILD register returns zero.

**Table 39-7 MCIP\_GFRC\_BUILD Field Description**

Field	Bit	Description
VERSION	[7:0]	<p>Version number</p> <ul style="list-style-type: none"> <li>■ 0x1: First version</li> <li>■ 0x2 : Second version; Renamed the global real-time counter (GRTC) to global free running counter (GFRC).</li> <li>■ 0x3: Current Version; Added the following commands: CMD_GFRC_SET_CORE, CMD_GFRC_READ_CORE, and CMD_GFRC_READ_HALT.</li> </ul>

### 39.5.6 Inter-Core Interrupt Unit Build Configuration Register, MCIP\_ICI\_BUILD

Address: 0xE0

Access: R

**Figure 39-9 MCIP\_ICI\_BUILD Register**



The MCIP\_ICI\_BUILD register is an optional BCR. If the inter-core interrupt unit is not configured in ARConnect, reading the MCIP\_ICI\_BUILD register returns zero.

**Table 39-8 MCIP\_ICI\_BUILD Field Description**

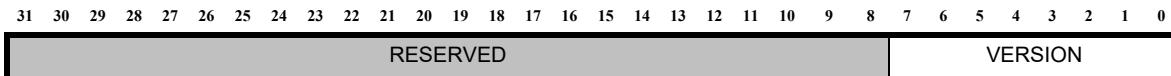
Field	Bit	Description
VERSION	[7:0]	Version number 0x2 : Current version. This register did not exist in earlier versions of ARConnect (version 0x1).

### 39.5.7 Inter-Core Debug Unit Build Configuration Register, MCIP\_ICD\_BUILD

Address: 0xE1

Access: R

**Figure 39-10 MCIP\_ICD\_BUILD Register**



The MCIP\_ICD\_BUILD register is an optional BCR. If the inter-core debug unit is not configured in ARConnect, reading the MCIP\_ICD\_BUILD register returns zero.

**Table 39-9 MCIP\_ICD\_BUILD Field Description**

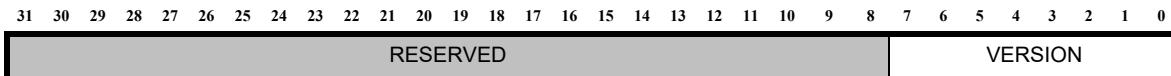
Field	Bit	Description
VERSION	[7:0]	Version number 0x2 : Current version. This register did not exist in earlier versions of ARConnect.

### 39.5.8 ARConnect Power Domain Management Build Configuration Register, MCIP\_PDM\_BUILD

Address: 0xE3

Access: R

**Figure 39-11 MCIP\_PDM\_BUILD Register**



The MCIP\_PDM\_BUILD register is an optional BCR. If the power-domain management unit is not configured in ARConnect, reading the MCIP\_PDM\_BUILD register returns zero.

**Table 39-10 MCIP\_PDM\_BUILD Field Description**

Field	Bit	Description
VERSION	[7:0]	Version number 0x1: Current version

## 39.6 Programming Restrictions

### 39.6.1 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.

### 39.6.2 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. Hence, 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.



# 40

## 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.

### 40.1 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.

### 40.2 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 40-1](#) lists the DMAC auxiliary registers.

**Table 40-1 DMA Auxiliary Registers**

Address	Name	Description
0x680	<a href="#">DMA Controller Configuration Register, DMACTRL</a>	DMA controller control register.
0x681	<a href="#">DMA Channel Enable Register, DMACENB</a>	DMA channel enable register

**Table 40-1 DMA Auxiliary Registers**

<b>Address</b>	<b>Name</b>	<b>Description</b>
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
0x690	DMA Channel Control Register, DMACTRL0 ( <a href="#">DMA Channel Control Register, DMACTRLx</a> )	Auxiliary DMA channel0 control register
0x691	DMA Channel Source Address Register, DMASARx ( <a href="#">DMA Channel Source Address Register, DMASARx</a> )	Auxiliary DMA channel0 source address register
0x692	DMA Channel Destination Address Register, DMADARx ( <a href="#">DMA Channel Destination Address Register, DMADARx</a> )	Auxiliary DMA channel destination0 address register
N/A (memory- mapped address)	DMA Channel Linked-List Pointer Register, DMALLPx	DMA channel structure pointer register

### 40.2.1 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 40-2](#) lists the registers that are accessible in the XY DMA controller.



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 40-2 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
0x690	DMA Channel Control Register, DMACTRL0 ( <a href="#">DMA Channel Control Register, DMACTRLx</a> )	Auxiliary DMA channel0 control register
0x691	DMA Channel Source Address Register, DMASARx ( <a href="#">DMA Channel Source Address Register, DMASARx</a> )	Auxiliary DMA channel0 source address register
0x692	DMA Channel Destination Address Register, DMADARx ( <a href="#">DMA Channel Destination Address Register, DMADARx</a> )	Auxiliary DMA channel destination0 address register

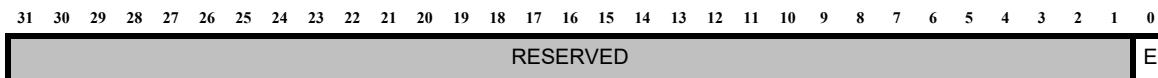
## 40.2.2 DMA Controller Configuration Register, DMACTRL

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 40-1 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 40-3 DMACTRL Field Description**

Field	Bit	Description
E	0	Enable or disable the DMA controller ■ 1: Enable ■ 0: Disable

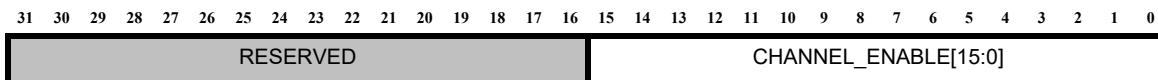
### 40.2.3 DMA Channel Enable Register, DMACENB

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 40-2 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 40-4 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

#### 40.2.4 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 40-3 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 40-5 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

### 40.2.5 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 40-4 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 40-6 DMACHPRI Field Description**

Field	Bit	Description
CHANNLE_PRIORITY_LEVEL_HIGH	[15:0]	1: Set the priority level of a DMA channel to high; each bit in this field [15:0] corresponds to a DMA channel

#### 40.2.6 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 40-5 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 40-7 DMACNPRI Field Description**

Field	Bit	Description
CHANNEL_PRIORITY_LEVEL_NORMAL	[15:0]	1: 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.

### 40.2.7 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 40-6 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 40-8 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.



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.

### 40.2.8 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 40-7 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 40-9 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. <ul style="list-style-type: none"> <li>■ 1: channel is active; data transfer is in progress</li> <li>■ 0: channel is inactive</li> </ul>

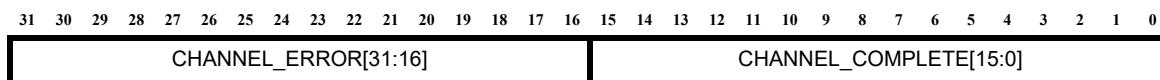
### 40.2.9 DMA Channel Status Register, DMACSTAT1

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 40-8 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 DMASTAT1 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 15. 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 40-10 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. <ul style="list-style-type: none"> <li>■ 1: Data transfer is complete</li> <li>■ 0: Channel is busy or inactive</li> </ul>

**Table 40-10 DMACSTAT1 Field Description**

Field	Bit	Description
CHANNEL_ERROR	[31:16]	<p>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:</p> <ul style="list-style-type: none"><li>■ 1: Error during data transfer</li><li>■ 0: No error during data transfer</li></ul> <p>Errors can occur due to the following reasons:</p> <ul style="list-style-type: none"><li>■ Auxiliary read or write to an undefined location</li><li>■ Auxiliary read from a write-only location</li><li>■ Auxiliary write to a read-only location</li><li>■ Memory read or write bus error</li><li>■ Data transfer that actively crosses a memory target boundary</li></ul>

### 40.2.10 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 40-9 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 40-11 DMACIRQ Field Description**

Field	Bit	Description
INTERRUPT_STATUS	[15:0]	<p>Read the interrupt status of each channel; each bit in this field [15:0] corresponds to a DMA channel.</p> <ul style="list-style-type: none"> <li>■ 1: the corresponding DMA channel has raised an interrupt</li> <li>■ 0: the corresponding DMA channel has not raised an interrupt</li> </ul>

#### 40.2.11 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.

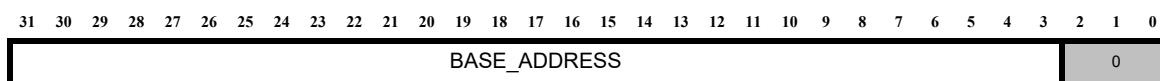


**Note** When `-sec_modes_option==true`, the DMA descriptors must be in a secure memory region.

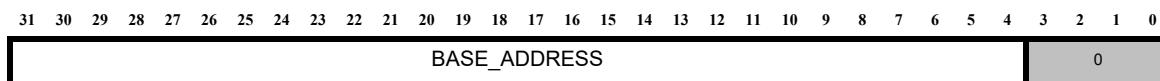
Depending on how many DMA channels are defined, the starting address table must be aligned accordingly. This eliminates the need to allocate the maximum memory table size for a DMA controller containing 16 channels.

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.

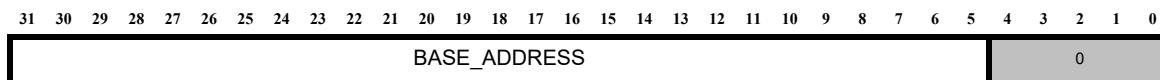
**Figure 40-10 DMACBASE Register for Two DMA Channels**

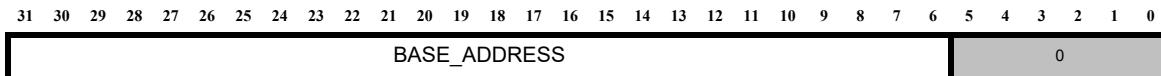


**Figure 40-11 DMACBASE Register for Three or Four DMA Channels**



**Figure 40-12 DMACBASE Register for Five to Eight DMA Channels**



**Figure 40-13 DMACBASE Register for 16 DMA Channels**

[Table 40-12](#) lists the DMA channel-specific register addresses when all the channel registers are mapped to the memory.

**Table 40-12 Memory-Mapped DMA Channel-Specific Registers**

Address	DMA Channel Descriptor	Description
DMACBASE+0x0	DSCPTR0	DSCPTR $x$ defines the address location of the DMA channel $x$ specific registers: <ul style="list-style-type: none"> <li>■ DSCPTR<math>x</math>: <i>DMACTRL<math>x</math>,</i></li> <li>■ DSCPTR<math>x+0x4</math>: <i>DMASAR<math>x</math>,</i></li> <li>■ DSCPTR<math>x+0x8</math>: <i>DMADAR<math>x</math></i></li> <li>■ DSCPTR<math>x+0xC</math>: <i>DMALLP<math>x</math></i></li> </ul>
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 40-13](#) 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 40-13 DMAC Channel Auxiliary Registers**

<b>Address</b>	<b>DMA Channel Data Structure</b>
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 40-14](#) lists the DMA channel-specific registers for a four-channel DMA controller with two memory-based channels and two auxiliary-based channels.

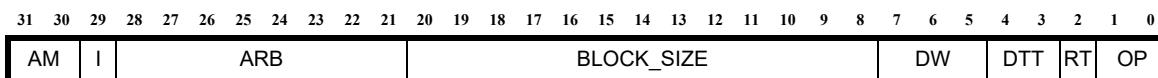
**Table 40-14 DMAC Channel Registers**

<b>Address</b>	<b>DMA Channel Data Structure</b>
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>

#### 40.2.12 DMA Channel Control Register, DMACTRL $x$

Address if memory-mapped:	See <a href="#">Table 40-12</a>
Address if auxiliary-mapped:	See <a href="#">Table 40-13</a>
Address if some DMA channels are auxiliary-mapped and some are memory-mapped:	See <a href="#">Table 40-14</a>
Access:	<ul style="list-style-type: none"><li>■ When SecureShield 2+2 mode is not configured: RW</li><li>■ When SecureShield 2+2 mode is configured: RW in the secure mode only</li></ul>
Reset:	0x00000000

**Figure 40-14 DMACTRLx Register**



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 40-15 DMACTRLx OP Field Description**

Field	Bit	Description
OP	[1:0]	<ul style="list-style-type: none"> <li>■ 00 : Invalid transfer</li> <li>■ 01 : Single transfer</li> <li>■ 10 : Linked-list transfer (auto-request)</li> <li>■ 11 : Linked-list transfer (manual-request)</li> </ul>

- **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 40-1](#) and [Example 40-2](#)
- **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 40-1](#) and [Example 40-2](#).

## 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 40-16 DMACTRLx RT Field Description**

Field	Bit	Description
RT	[2]	<ul style="list-style-type: none"> <li>■ 0 : 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.</li> <li>■ 1 : 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.           <ul style="list-style-type: none"> <li>- 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.</li> </ul> </li> </ul>

## DTT

This field is used to define the source and destination target types as: auxiliary or memory.

**Table 40-17 DMACTRLx DTT Field Description**

Field	Bit	Description
DTT	[4:3]	<ul style="list-style-type: none"> <li>■ 00 : Memory to memory</li> <li>■ 01 : Memory to auxiliary</li> <li>■ 10 : Auxiliary to memory</li> <li>■ 11 : Auxiliary to auxiliary</li> </ul>

- **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 40-18 DMACTRLx DW Field Description**

Field	Bit	Description
DW	[7:5]	<ul style="list-style-type: none"> <li>■ 000: Data width is a byte and address increment is a byte</li> <li>■ 001: Data width is a byte and address increment is 16-bits</li> <li>■ 010: Data width is a byte and address increment is 32-bits</li> <li>■ 011: Data width is 16-bits and address increment is 16-bits</li> <li>■ 100: Data width is 16-bits and address increment is 32-bits</li> <li>■ 101: Data width is 32-bits and address increment is 32-bits</li> <li>■ 110: 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.</li> </ul>

## 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 40-19 DMACTRLx BLOCK\_SIZE Field Description**

Field	Bit	Description
BLOCK_SIZE	[20:8]	<ul style="list-style-type: none"> <li>■ 0000000000000: one byte</li> <li>■ 0000000000001: two bytes</li> <li>■ 0000000000010: three bytes</li> <li>■ ....</li> <li>■ 1111111111111: 8 Kbytes</li> </ul>

## 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 40-20 DMACTRLx ARB Field Description**

Field	Bit	Description
ARB	[28:21]	<ul style="list-style-type: none"> <li>■ 0x00: No arbitration</li> <li>■ 0x01: Arbitrate after 1 DMA cell transfer</li> <li>■ 0x02: Arbitrate after 2 DMA cell transfers</li> <li>■ 0x03: Arbitrate after 3 DMA cell transfers</li> <li>■ ....</li> <li>■ 0xFF: Arbitrate after 255 DMA cell transfers</li> </ul>

## I

The interrupt mode bit defines whether an internal or external interrupt is raised when a block transfer is complete.

**Table 40-21 DMACTRLx I Bit Description**

Field	Bit	Description
I	29	<ul style="list-style-type: none"> <li>■ 0: interrupt mode is disabled; interrupts are not raised when a DMA data transfer is complete.</li> <li>■ 1: interrupt mode is enabled; interrupts are raised when a DMA data transfer is complete.</li> </ul>

## 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 40-22](#) lists the configurations available.

**Table 40-22 DMACTRLx AM Field Description**

Field	Bit	Description
AM	[31:30]	<ul style="list-style-type: none"> <li>■ 00: Source and address are not incremented</li> <li>■ 01: Source address is incremented, destination address is not incremented</li> <li>■ 10: Source address is not incremented, destination address is incremented</li> <li>■ 11: Source address is incremented and the destination address is incremented</li> </ul>

## Reserved fields in DMACTRLx

The DMACTRLx register includes the following reserved fields and should not be used.

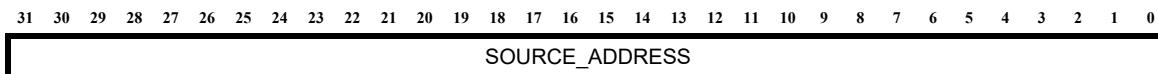
1. For XY DMA controller, the DMACTRLx [1 : 0] == 2 | 3 values are reserved. You cannot perform linked-list transfers with a XY DMA controller. If you do linked-list transfer with a XY DMA controller, the request is ignored and the respective channel is disabled; no error is raised.
2. Regardless of 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.

### 40.2.13 DMA Channel Source Address Register, DMASARx

Address if memory-mapped:	See <a href="#">Table 40-12</a>
Address if auxiliary-mapped:	See <a href="#">Table 40-13</a>
Address if some DMA channels are auxiliary-mapped and some are memory-mapped:	See <a href="#">Table 40-14</a>
Access:	<ul style="list-style-type: none"> <li>■ When SecureShield 2+2 mode is not configured: RW</li> <li>■ When SecureShield 2+2 mode is configured: RW in the secure mode only</li> </ul>
Reset:	0x00000000

**Figure 40-15 DMASARx Register**



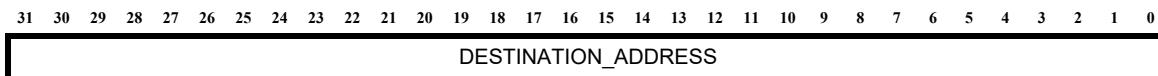
This register provides the end address location of the source memory target. The address is specified in bytes for a memory target or 32-bit data locations for an auxiliary space. The transfer block size field in the DMACTRLx register is subtracted from this value to calculate the true starting address 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.

#### 40.2.14 DMA Channel Destination Address Register, DMADAR<sub>x</sub>

Address if memory-mapped:	See <a href="#">Table 40-12</a>
Address if auxiliary-mapped:	See <a href="#">Table 40-13</a>
Address if some DMA channels are auxiliary-mapped and some are memory-mapped:	See <a href="#">Table 40-14</a>
Access:	<ul style="list-style-type: none"> <li>■ When SecureShield 2+2 mode is not configured: RW</li> <li>■ When SecureShield 2+2 mode is configured: RW in the secure mode only</li> </ul>
Reset:	0x00000000

**Figure 40-16 DMADAR<sub>x</sub> 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 DMACTRL<sub>x</sub> 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)  
}
```

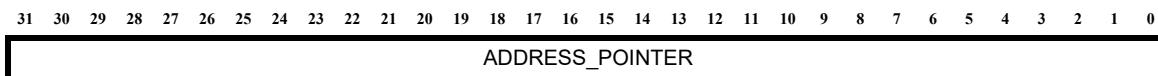
The following pseudo code is used to calculate the end auxiliary address when only the burst size and required starting address are known.

```
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)  
}
```

#### 40.2.15 DMA Channel Linked-List Pointer Register, DMALLPx

Address if memory-mapped:	See <a href="#">Table 40-12</a>
Address if auxiliary-mapped:	See <a href="#">Table 40-13</a>
Access:	<ul style="list-style-type: none"> <li>▪ When SecureShield 2+2 mode is not configured: RW</li> <li>▪ When SecureShield 2+2 mode is configured: RW in the secure mode only</li> </ul>
Reset:	0x00000000

**Figure 40-17 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.

#### Example 40-1 Linked-list Transfers for Memory-Mapped DMA Channels

[Table 40-27](#) and table [Table 40-27](#) are used to illustrate the linked-list transfers for memory-based DMA channels.

**Table 40-23 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 40-24 Initial State of DMA Channel Descriptor Registers for Example 40-1**

Memory Address	Value
0xC0000	0x00C2FF07
0xC0004	0x80000
0xC0008	0x80000

**Table 40-24 Initial State of DMA Channel Descriptor Registers for Example 40-1**

Memory Address	Value
0xC000C	0xB0000

- 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 (DMACTRL0), 0xC0004 (DMASAR0), and 0xC0008 (DMADAR0) are read and the defined DMA transfer executed.
- During arbitration (where the state of DMACTRLx must be saved) the location in the address table is read again to locate the address location of DMACTRL0. The memory location is subsequently updated with the latest information for DMACTRL0. The loading of the DMA descriptor and the updating of the DMACTRLx register during arbitration continues until the block data transfer completes.
- When the block transfer completes, DMACTRL0 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 40-25 DMAC Channel Descriptors**

Channel	System Memory Address Location	Descriptor Pointer Address
DMA Channel 0	DMACBASE	0xB0000; see <a href="#">Table 40-24</a>
DMA Channel 1	DMACBASE+0x4	0x60000 (DSCPTR1)

**Table 40-26 Final State of DMA Channel Descriptor Registers for Example 40-2**

Memory Address	Value
0xC0000	0x00C2FF04
0xC0004	0x80000
0xC0008	0x90000
0xC000C	0xB0000
0xB0000	0x00C2FF04
0xB0004	0xA0000
0xB0008	0xB0000

**Table 40-26 Final State of DMA Channel Descriptor Registers for Example 40-2**

Memory Address	Value
0xB000C	0xC0000

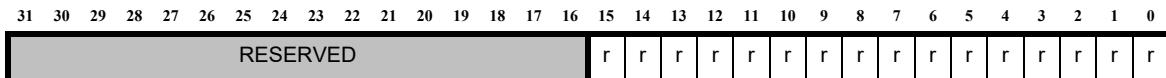
### 40.2.16 DMA Channel Reset Register, DMACRST

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 40-18 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 bit remains set until the DMA channel is reset.



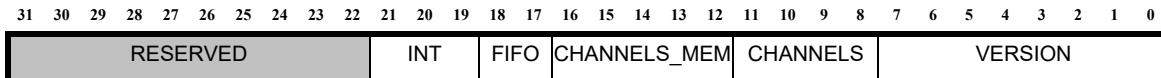
**Note** A software or peripheral request that is made to a DMA channel that is being reset is always ignored.

#### 40.2.17 DMA Build Configuration Register, DMAC\_BUILD

Address: 0xCD

Access: R

**Figure 40-19 DMAC\_BUILD Register**



This register indicates the presence of the DMA controller and its configuration.

**Table 40-27 DMAC\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>Version of the DMA controller</b> <ul style="list-style-type: none"> <li>■ 0x01: Initial version</li> <li>■ 0x02: Current version</li> </ul>
CHANNELS	[11:8]	<b>Number of DMA channels</b> <ul style="list-style-type: none"> <li>■ 0x0: 1</li> <li>■ 0x1: 2</li> <li>■ 0x2: 3</li> <li>■ ...</li> <li>■ ...</li> <li>■ 0xF: 16</li> </ul>
CHANNELS_MEM	[16:12]	<b>Defines the number of DMA channels that have their registers mapped to the auxiliary space</b> <ul style="list-style-type: none"> <li>■ 0x0: All DMA channels have memory-mapped registers</li> <li>■ 0x1: 1 DMA channel</li> <li>■ 0x2: 2 DMA channels</li> <li>■ ...</li> <li>■ ....</li> <li>■ 0xF: 15 DMA channels</li> <li>■ 0x10: 16 DMA channels</li> </ul>
FIFO	[18:17]	<b>Depth of the internal FIFO</b> <ul style="list-style-type: none"> <li>■ 00: One byte</li> <li>■ 01: Two bytes</li> <li>■ 10: Reserved</li> <li>■ 11: Four bytes</li> </ul>

**Table 40-27 DMAC\_BUILD Field Descriptions (Continued)**

Field	Bit	Description
INT	[21:29]	<b>Interrupt configuration of the DMA controller</b> <ul style="list-style-type: none"><li>■ 000: None</li><li>■ 001: Single-Internal</li><li>■ 010: Multiple-Internal</li><li>■ 011: Single-External</li><li>■ 100: Multiple-External</li></ul>



# Part 4

## ARCv2 Scalar DSP Extensions

### In this part:

- [Scalar DSP Introduction](#)
- [Data Organization and Addressing](#)
- [DSP Auxiliary Interface](#)
- [Scalar DSP Instruction Set Details](#)



# 41

## 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  $32 \times 32$  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.

### 41.1 Key Features of the ARCv2 ISA DSP Extensions

- Basic Saturating Arithmetic instructions
- Vector Unpacking instructions
- Vector ALU instructions
- Accumulator instructions
- Vector SIMD 16x16 MAC instructions
- Dual Inner Product 16x16 MAC instructions
- 32x32 and 16x16 MAC instructions
- Dual 16x8 MAC instructions
- Single-cycle Complex multiply, conjugated multiply, and FFT instructions
- 32x16 MAC instructions
- Accumulator shifting allowed in the [-72:72] range.
- A small address generation unit (AGU)

## 41.2 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.

## 41.3 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. 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 41-1](#) describes the configuration options for the programming model supported by the ARCv2 architecture.

### 41.3.1 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 41-1](#).

**Table 41-1 ARCv2 Programming Model Configuration Options**

Configuration Option	Value Range	Description
-agu_size	small, medium, large	Predefined configurations of the address-pointer registers, modifier registers, and offset registers for address generator unit based XY accesses.
-dsp_impl	inferred, optimized	Indicates the DSP hardware that is included. <ul style="list-style-type: none"> <li>■ inferred: indicates that the DSP hardware is inferred from the Verilog code.</li> <li>■ optimized: indicates that optimized DSP hardware is used that includes carry-save components.</li> </ul>
-dsp_itu	false, true	Indicates support for preaccumulate hardware shifting required by the DSP ITU codecs. <ul style="list-style-type: none"> <li>■ false: preaccumulate shifting hardware is not supported.</li> <li>■ true: preaccumulate shifting hardware is supported.</li> </ul>
-dsp_divsqrt	none, radix2, radix4	Enables support for the divide and square-root DSP instructions. <ul style="list-style-type: none"> <li>■ none: DIV/REM, and square root DSP instructions are not supported.</li> <li>■ radix2: Radix-2 bit-serial DIV, DIVU, REM, REMU, and square root instructions are supported.</li> <li>■ radix4: Radix-4 fast DIV, DIVU, REM, REMU, and square root instructions are supported.</li> </ul>
-dsp_wide	false, true	Indicates support for the 64-bit operands in the DSP instructions. <ul style="list-style-type: none"> <li>■ false: 64-bit operands for the DSP instructions are not supported.</li> <li>■ true: 64-bit operands are supported with the DSP instructions.</li> </ul>

### 41.3.2 ISA Options

**Table 41-2** 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.

**Table 41-2 ARCv2 Instruction Set Options**

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-has_dsp	true	Instructions documented in “ <a href="#">Scalar DSP Instruction Set Details</a> ” except the following: DMPYWHF, DMACWHF, QMPYHF, QMACHF, VADDS4H, VSUBS4H, VADDSUBS4H, VSUBADDS4H, VADDS2, VSUBS2, VADDSUBS, VSUBADDS	
-dsp_wide	true	DMPYWHF, DMACWHF, QMPYHF, QMACHF, VADDS4H, VSUBS4H, VADDSUBS4H, VSUBADDS4H, VADDS2, VSUBS2, VADDSUBS, VSUBADDS	

# 42

## Data Organization and Addressing

This chapter describes the data organization and addressing used by the ARCv2-based DSP extensions.

### 42.1 DSP Data Formats

The set of DSP extension instructions, enabled by the HAS\_DSP option, use the following data formats.

#### 42.1.1 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 42-1 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																															

#### 42.1.2 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 42-2 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

#### 42.1.3 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 42-3 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															

#### 42.1.4 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 42.1.7. 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, e.g. MPYWHFL.
- Instructions whose mnemonic has suffix 'M', take a Q15 scalar operand from the most significant 16 bits of a 32-bit operand, e.g. MSUBWHFM.

The maximum representable value for a 16-bit signed fraction is 0.999969 and minimum value is -1.0.

#### 42.1.5 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 42-4 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															

#### 42.1.6 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 42-5 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															

#### 42.1.7 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 42-6 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																S																	

#### 42.1.8 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 42-7 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																S																	

#### 42.1.9 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 42-8 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																																

#### 42.1.10 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 42-9 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																															

#### 42.1.11 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 42-10 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																														

## 42.1.12 Accumulators

The DSP component uses the following auxiliary registers for DSP operations:

- “ACC0\_LO” on page 1229
- “ACC0\_HI” on page 1231
- “ACC0\_GLO” on page 1230
- “ACC0\_GHI” on page 1232

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.

### 42.1.12.1 Vector Accumulators

The ACC0\_LO, ACC0\_HI, ACC0\_GLO and ACC0\_GHI 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:

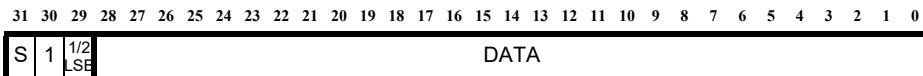
Operation	ACC0_GHI GUARD	ACC0_HI	ACC0_GLO GUARD	ACC0_LO
2x32 bits		val1[31:0]		val0[31:0]
1x64 bits		val[63:32]		val[31:0]
1x32b		val[31:0]		
1x40b	val[39:32]	val[31:0]		
2x40b	val1[39:32]	val1[31:0]	val0[39:32]	val0[31:0]



The 1x72 bit operation is explained in section “Wide Accumulator” on page 1223.

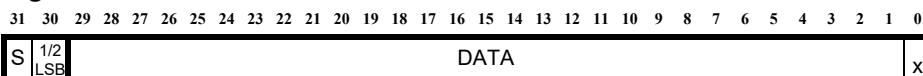
When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==0`, Figure 42-1 represents the vector accumulators (`ACC0_LO` and `ACC0_HI`).

**Figure 42-1 Vector Accumulator When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==0`**



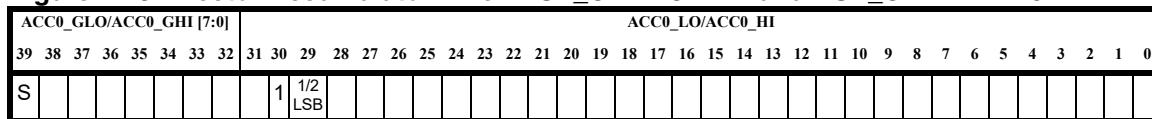
When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`, Figure 42-2 represents the vector accumulators (`ACC0_LO` and `ACC0_HI`). In this mode, the LSB is set only on saturation.

**Figure 42-2 Vector Accumulator When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`**



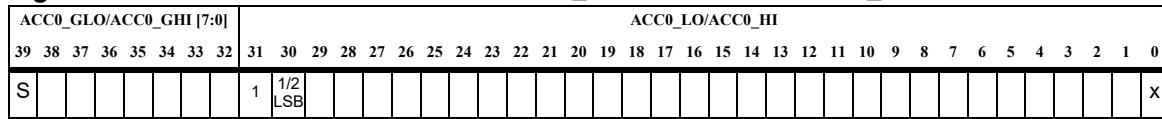
When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==0`, Figure 42-3 represents the vector accumulators.

**Figure 42-3 Vector Accumulator When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==0`**



When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`, Figure 42-4 represents the vector accumulators. In this mode, the LSB is set only on saturation.

**Figure 42-4 Vector Accumulator When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`**



## 42.1.12.2 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:

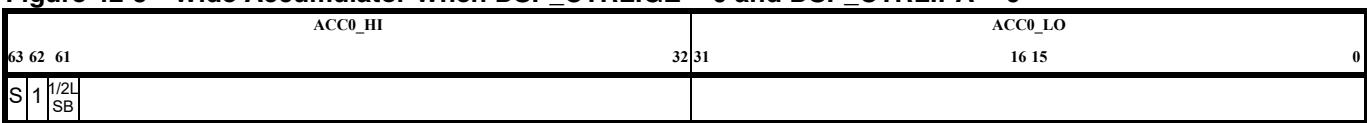
Operation	ACC0_GHI GUARD	ACC0_HI	ACC0_LO
1x72 bits	val[71:64]	val[63:32]	val[31:0]

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 42-5 represents the wide accumulator.

**Figure 42-5 Wide Accumulator When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==0`**



When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`, Figure 42-6 represents the wide accumulator. In this mode, the LSB is set only on saturation.

**Figure 42-6 Wide Accumulator When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`**

ACC0_HI				ACC0_LO	
63 62 61				32	31
S   1/2L SB					X

When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==0`, Figure 42-7 represents the wide accumulator.

**Figure 42-7 Wide Accumulator When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==0`**

ACC0_GHI [7:0]		ACC0_HI				ACC0_LO	
71	63 62 61	32	31	16	15	0	
S		1	1/2L SB				X

When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`, Figure 42-8 represents the wide accumulator.

**Figure 42-8 Wide Accumulator When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`**

ACC0_GHI [7:0]		ACC0_HI				ACC0_LO	
71	63 62 61	32	31	16	15	0	
S		1	1/2L SB				X

## 42.1.13 Accumulator Saturation

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.

## 42.1.14 Accumulator Operations

Table 42-11 provides the list of instructions that you can use to shift, set, read, divide, and compute the square root of the accumulators:

**Table 42-11 Accumulator Instructions**

	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 42-11 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



# 43

## DSP Auxiliary Interface

### 43.1 Extension Core Registers

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

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.

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 43-1 ACCL for fusedMultiplyAdd and fusedMultiplySubtract Operations**

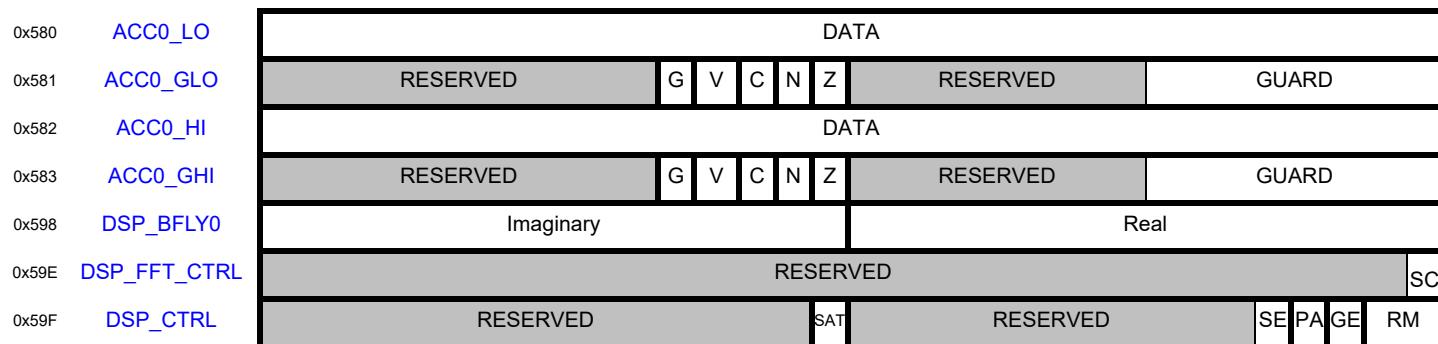
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	Exponent																														



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.

## 43.2 Auxiliary Register Set

**Figure 43-2 Auxiliary Register Map**



### 43.2.1 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 43-1 DSP Auxiliary Registers**

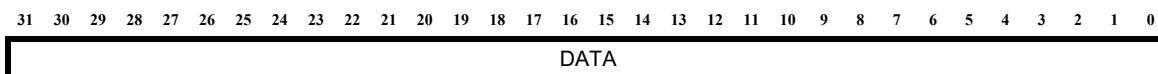
Address	Auxiliary Register Name	Description
0x580	Accumulator Low Register, ACC0_LO	Low accumulator register.
0x581	Low Accumulator Guard and Status Register, ACC0_GLO	Low accumulator guard and status register.
0x582	Accumulator High Register, ACC0_HI	High accumulator register.
0x583	High Accumulator 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.

### 43.2.2 Accumulator Low Register, ACC0\_LO

Address: 0x580

Access: rw

**Figure 43-3 ACC0\_LO Register**



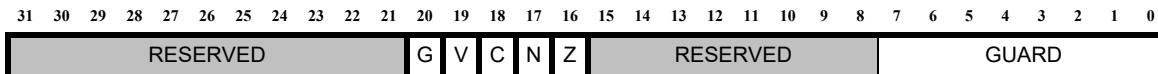
This register holds the lower 32 bits of the result from a vector accumulate operation. For complex math operations, this register holds the real part of a complex operand in a little-endian mode and the imaginary part of the complex operand in a big-endian mode. For operations that involve a wide accumulator, this register holds the least significant 32 bits of the wide accumulator.

### 43.2.3 Low Accumulator Guard and Status Register, ACC0\_GLO

Address: 0x581

Access: rw

**Figure 43-4 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 43-2 ACC0\_GLO Field Descriptions**

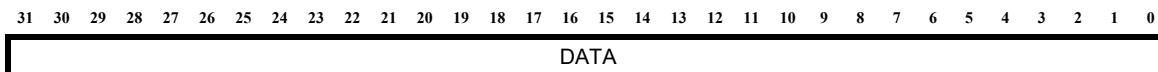
Field	Bit	Description
GUARD	[7:0]	Guard bits; overflow bits of the low vector accumulator. For example, if a the low vector accumulator (ACC0_LO) overflows, this field stores the overflow bits (up to 8 bits). This field is valid only if DSP_CTRL.GE==1.
Z	[16]	Zero flag; this bit is set only by the fractional 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).

#### 43.2.4 Accumulator High Register, ACC0\_HI

Address: 0x582

Access: rw

**Figure 43-5 ACC0\_HI Register**



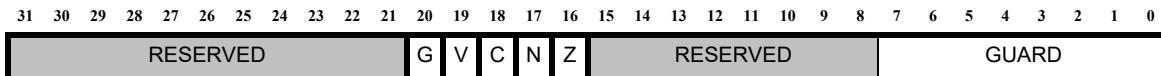
This register holds the higher 32 bits of the result from a vector accumulate operation. For complex math operations, this register holds the imaginary part of a complex operand in a little-endian mode and the real part of the complex operand in a big-endian mode. For operations that involve a wide accumulator, this register holds the most significant 32 bits of the wide accumulator.

### 43.2.5 High Accumulator Guard and Status Register, ACC0\_GHI

Address: 0x583

Access: rw

**Figure 43-6 ACC0\_GHI Register**



This register holds the guard bits and the status flags for the ACC0\_HI accumulator.

**Table 43-3 ACC0\_GHI Field Descriptions**

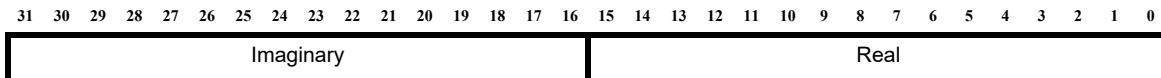
Field	Bit	Description
GUARD	[7:0]	Guard bits; overflow bits of the high vector accumulator. For example, if a the low vector accumulator (ACC0_HI) overflows, this field stores the overflow bits (up to 8 bits). This field is valid only if DSP_CTRL.GE==1.
Z	[16]	Zero flag; this bit is set only by the fractional 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 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_HI).

### 43.2.6 DSP Butterfly Instructions Data Register, DSP\_BFLY0

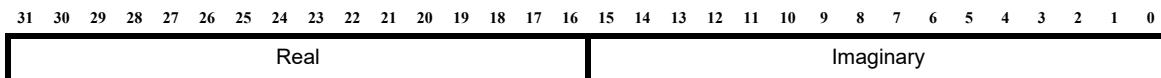
Address: 0x598

Access: rw

**Figure 43-7 DSP\_BFLY0 Register in Little-Endian Mode**



**Figure 43-8 DSP\_BFLY0 Register in Big-Endian Mode**



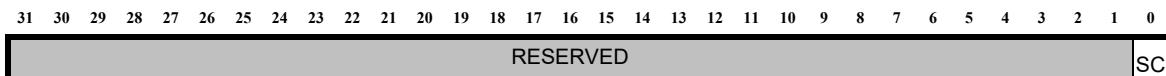
This register stores the implicit complex A0 operand that is used in the butterfly instructions. The DSP\_BFLY0 register consists of the real and imaginary components of a 16+16 bit complex fractional number. The CBFLYH1R instruction can modify this register. The CBFLYH0R and CBFLYH1R instructions can implicitly read this register. This register is present in the processor build only if `dsp_complex==true`.

### 43.2.7 DSP FFT Control Register, DSP\_FFT\_CTRL

Address: 0x59E

Access: rw

**Figure 43-9 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 43-4 DSP\_FFT\_CTRL Field Descriptions**

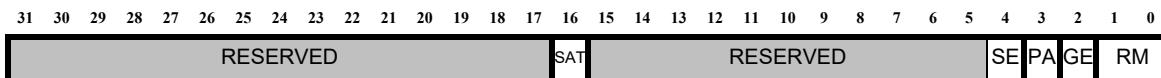
Field	Bit	Description
SC	[0]	<b>Controls the scaling of the complex results</b> 0: no scaling 1: the butterfly instructions (CBFLYHF0R and CBFLYH1R) scale the complex result by right-shifting the result by one bit.

### 43.2.8 DSP Control Register, DSP\_CTRL

Address: 0x59F

Access: rw

**Figure 43-10 DSP\_CTRL Register**



This register controls the rounding and guard bit behavior of the DSP multiply and accumulate operations.

**Table 43-5 DSP\_CTRL Field Descriptions**

Field	Bit	Description
RM	[1:0]	<b>Indicates the rounding mode</b> If <code>dsp_accshift==full</code> <ul style="list-style-type: none"> <li>■ 0: no rounding; round down to the nearest representable number</li> <li>■ 1: truncation; round down to the nearest representable number</li> <li>■ 2: round up; add 1/2 LSB to the result and truncate</li> <li>■ 3: 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.</li> </ul> If <code>dsp_accshift==limited</code> <ul style="list-style-type: none"> <li>■ RM [0] == 0.</li> <li>■ 0: no rounding; round down to the nearest representable number</li> <li>■ 2: round up; add 1/2 LSB to the result and truncate</li> </ul>
GE	[2]	<b>Enable guard bits</b> 0: disable guard bits; the wide accumulator saturates at 64 bits and the vector accumulators saturate at 32 bits 1: enable guard bits; the wide accumulator saturates at 72 bits and the vector accumulators saturate at 40 bits

**Table 43-5 DSP\_CTRL Field Descriptions (Continued)**

Field	Bit	Description
PA	[3]	<p><b>Enable pre-accumulation shift mode</b></p> <p>If <code>dsp_itu==true</code></p> <ul style="list-style-type: none"> <li>■ 0: post accumulation. The fractional product is not shifted before accumulation. The accumulator is of type 1Q30.</li> <li>■ 1: 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>.</li> </ul> <p>If <code>dsp_itu==false</code></p> <ul style="list-style-type: none"> <li>■ The PA bit is fixed to 0; pre-accumulation shift mode is not supported.</li> </ul>
SE	[4]	<p><b>Enable saturation for fractional instructions</b></p> <ul style="list-style-type: none"> <li>■ 0: disable saturation for guarded operations with fractional instructions</li> <li>■ 1: enable saturation for guarded operations with fractional instructions</li> </ul>
SAT	[16]	<p><b>Sticky saturation status</b></p> <p>1: indicates that a DSP operation is saturated in one of the following cases:</p> <ul style="list-style-type: none"> <li>■ Accumulator saturation</li> <li>■ Result saturation</li> <li>■ Multiply pre-accumulate saturation when <code>DSP_CTRL.GE==0</code> and <code>DSP_CTRL.PA==1</code></li> </ul> <p>This bit is a sticky flag. You must explicitly write a 0 to this bit to clear this flag.</p>

### 43.3 Build Configuration Registers

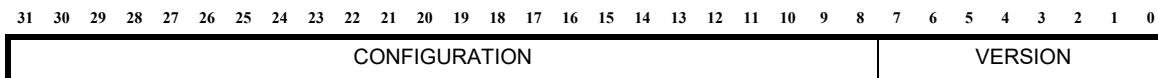
The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCv2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

### 43.3.1 DSP Build Configuration Register, DSP\_BUILD

Address: 0x7A

Access: R

**Figure 43-11 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 43-6 DSP\_BUILD Field Descriptions**

Field	Bit	Description
VERSION	[7:0]	<b>DSP version number</b> 0x21: current version
CONFIGURATION	[9:8]	<b>Indicates support for the divide and square-root instructions</b> <ul style="list-style-type: none"> <li>■ 0— DIV/REM instructions are not supported</li> <li>■ 1— Radix-2 bit-serial DIV, DIVU, REM and REMU implementation</li> <li>■ 2— Radix-4 fast DIV, DIVU, REM and REMU implementation</li> </ul>
CONFIGURATION	[10]	<b>Indicates support for the single-cycle complex instructions</b> Indicates support for single-cycle complex hardware. <ul style="list-style-type: none"> <li>■ 0— Support for two-cycle complex MAC hardware; this option does not include the complex butterfly hardware</li> <li>■ 1— Support for single-cycle complex MAC, conjugated multiply, and FFT butterfly hardware</li> </ul>
CONFIGURATION	[12:11]	<b>Indicates support for the accumulator shift</b> <ul style="list-style-type: none"> <li>■ 1— Shift accumulator in the range [-8:8]</li> <li>■ 2— Shift accumulator in the range [-72:72]</li> </ul>
CONFIGURATION	[13]	<b>Indicates support for the ITU pre-accumulate support.</b> <ul style="list-style-type: none"> <li>■ 0— Preaccumulate shifting hardware is not supported</li> <li>■ 1— Preaccumulate shifting hardware is supported</li> </ul>
CONFIGURATION	[31:14]	<b>Reserved. Read as 0.</b>



# 44

## Scalar DSP Instruction Set Details

### 44.1 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.

### 44.2 List of Instructions

Table 44-1 summarizes the 32-bit alongside the 16-bit instructions supported by the ARCv2 ISA.

**Table 44-1 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 44-1 List of Instructions (Continued)**

<b>32-Bit Instructions</b>	
<b>Instruction</b>	<b>Operation</b>
<b>ASRSR</b>	Shift right rounding and saturating if value is negative
<b>CBFLYHF0R</b>	16 bit + 16 bit FFT butterfly, first half; return first result
<b>CBFLYHF1R</b>	16 bit + 16 bit FFT butterfly, second half; return first result with rounding
<b>CMACCHFR</b>	16 bit + 16 bit complex signed fractional conjugated multiplication and accumulation with rounding
<b>CMACCHNFR</b>	16 bit + 16 bit complex signed fractional conjugated multiplication and accumulation with rounding, and returned without a fractional shift
<b>CMACHFR</b>	16 + 16 bit complex signed fractional multiplication and accumulation with rounding
<b>CMACHNFR</b>	16 bit + 16 bit complex signed fractional multiplication and accumulation with rounding
<b>CMPYCHFR</b>	16 bit + 16 bit complex signed fractional conjugated multiplication with rounding
<b>CMPYCHNFR</b>	16 bit + 16 bit complex signed fractional conjugated multiplication with rounding, and returned without a fractional shift
<b>CMPYHFMR</b>	16 + 16 bit complex signed fractional multiplication with rounding
<b>CMPYHNFR</b>	16 bit + 16 bit complex signed fractional multiplication with rounding
<b>DIVF</b>	Signed 32-bit fractional division
<b>DIVU</b>	Unsigned integer Divide
<b>DMACH</b>	Dual 16x16 multiply and accumulate
<b>DMACHBL</b>	Dual vector 16 bitx8 bit signed integer multiplication and accumulation; lower bytes
<b>DMACHBM</b>	Dual vector 16 bitx8 bit signed integer multiplication and accumulation; higher bytes
<b>DMACHF</b>	Dual vector 16 bit signed fractional multiplication and accumulation
<b>DMACHFR</b>	Dual vector 16 bit signed fractional rounding multiplication and accumulation
<b>DMACHU</b>	Dual unsigned 16x16 multiply and accumulate
<b>DMPYH</b>	Dual 16x16 multiplication
<b>DMPYHBL</b>	Dual vector 16 bitx8 bit signed integer multiplication; lower bytes
<b>DMPYHBM</b>	Dual vector 16 bitx8 bit signed integer multiplication; higher bytes
<b>DMPYHF</b>	Dual vector 16 bit signed fractional multiplication
<b>DMPYHFR</b>	Dual vector 16 bit signed fractional rounding multiplication
<b>DMPYHU</b>	Dual unsigned 16x16 multiplication

**Table 44-1 List of Instructions (Continued)**

32-Bit Instructions	
Instruction	Operation
DMPYHWF	Dual vector 16 bit signed fractional multiplication and 32-bit writeback
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)
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
MPYF	32 bit signed fractional saturating multiplication
MPYFR	32 bit signed fractional saturating an rounding multiplication

**Table 44-1 List of Instructions (Continued)**

<b>32-Bit Instructions</b>	
<b>Instruction</b>	<b>Operation</b>
<b>MPYMU</b>	32 x 32 Unsigned Multiply (msw)
<b>MPYHU</b>	
<b>MPYU</b>	32 x 32 Unsigned Multiply (lsw)
<b>MPYUW</b>	16 bit unsigned integer multiplication
<b>MPYUW_S</b>	16 bit unsigned integer multiplication
<b>MPYW</b>	16 X 16 signed multiply
<b>MPYW_S</b>	16 X 16 signed multiply
<b>MPYWHFL</b>	32x16 bit signed fractional multiplication, lower signed 16 bits
<b>MPYWHFLR</b>	32x16 bit signed fractional multiplication with rounding, lower signed 16 bits
<b>MPYWHFM</b>	32x16 bit signed fractional multiplication, higher signed 16 bits
<b>MPYWHFMR</b>	32x16 bit signed fractional multiplication with rounding, higher signed 16 bits
<b>MPYWHL</b>	32x16 bit signed multiplication, lower signed 16 bits
<b>MPYWHKL</b>	Signed 32x16 integer multiplication, and shift-right the lower bits
<b>MPYWHKUL</b>	Unsigned 32x16 integer multiplication, and shift-right the lower bits
<b>MPYWHUL</b>	32x16 bit unsigned multiplication, lower signed 16 bits
<b>MSUBDF</b>	32 bit signed fractional saturating subtraction; 64-bit result
<b>MSUBFR</b>	32 bit fractional saturating and rounding subtraction
<b>MSUBF</b>	32 bit fractional saturating subtraction
<b>MSUBWHFL</b>	Signed 32x16 fractional multiplication and subtraction
<b>MSUBWHFLR</b>	Signed 32x16 fractional multiplication and subtraction with rounded result
<b>MSUBWHFM</b>	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand
<b>MSUBWHFMR</b>	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
<b>NEGS</b>	Negative saturated 16-bit value
<b>NEGSH</b>	Negative saturated 32-bit value
<b>NORMACC</b>	Normalize the accumulator
<b>REM</b>	Signed Integer Remainder

**Table 44-1 List of Instructions (Continued)**

<b>32-Bit Instructions</b>	
<b>Instruction</b>	<b>Operation</b>
<b>REMU</b>	Unsigned Integer Remainder
<b>RNDH</b>	Rounding and saturating a 32-bit value to a 16-bit value
<b>SATH</b>	Saturating 32-bit value to a 16-bit value
<b>SATF</b>	Saturate a register based on the STATUS32.N and STATUS32.V flags
<b>SBCS</b>	Signed subtraction with saturation and carry in
<b>SBCS</b>	Subtract with carry
<b>SETACC</b>	Set the accumulator
<b>SQRT</b>	Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result
<b>SQRTF</b>	Compute the square-root of an unsigned Q31 fraction, yielding a Q31 fractional result
<b>SUBS</b>	Saturating 32-bit subtraction
<b>VABS2H</b>	Two way 16 bit vector absolute
<b>VABSS2H</b>	Two way 16 bit vector saturating absolute
<b>VADD2H</b>	Dual 16-bit vector addition
<b>VADD4B</b>	Four way add
<b>VADDS2H</b>	Two way vector saturating add 16 bits
<b>VADDSUB2H</b>	Dual 16-bit vector add and subtract
<b>VADDSUBS2H</b>	Two way vector saturating add subtract
<b>VALGN2H</b>	Two-way 16-bit vector align
<b>VASL2H</b>	Two way arithmetic shift left 16 bits
<b>VASLS2H</b>	Two way saturating arithmetic shift left 16 bits
<b>VASR2H</b>	Two way arithmetic shift right 16 bits
<b>VASRS2H</b>	Two way arithmetic saturating shift right 16 bits
<b>VASRSR2H</b>	Two way arithmetic saturating and rounding shift right 16 bits
<b>VEXT2BHL</b>	Vector extend the lower two bytes to the higher two bytes
<b>VEXT2BHL</b>	Compose a dual Q15 vector by expanding the lower two Q7 bytes of an operand
<b>VEXT2BHM</b>	Vector extend the higher two bytes to the lower two bytes
<b>VEXT2BHMF</b>	Compose a dual Q15 vector by expanding the higher two Q7 bytes of an operand

**Table 44-1 List of Instructions (Continued)**

32-Bit Instructions	
Instruction	Operation
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 writeback
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
VPACK2HBM	Pack the higher two bytes of a vector from the lower 8-bits of two 16-bit vectors of an operand

**Table 44-1 List of Instructions (Continued)**

<b>32-Bit Instructions</b>	
<b>Instruction</b>	<b>Operation</b>
<b>VPACK2HBMF</b>	Pack the higher two bytes of a vector from the higher 8-bits of two 16-bit vectors of an operand
<b>VPERM</b>	Perform byte permutation with zero or sign extension
<b>VREP2HL</b>	Replicate lower 16-bits into the upper 16-bits
<b>VREP2HM</b>	Replicate higher 16-bits to lower 16-bits
<b>VSEXT2BHL</b>	Vector sign extend the lower two bytes to the higher two bytes
<b>VSEXT2BHM</b>	Vector sign extend the higher two bytes to the lower two bytes
<b>VSUB4B</b>	Four way subtract
<b>VSUBS4H</b>	Four-way 16-bit vector addition.
<b>VSUB2H</b>	Dual 16-bit vector subtraction
<b>VSUBADD2H</b>	Dual 16-bit vector subtract and add
<b>VSUBADDS2H</b>	Two way vector saturating subtract add
<b>VSUBS2H</b>	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 = SAT32(ABS(c));`

### Instruction Format

`op b, c`

### Timing Characteristic

Issue one instruction per cycle; two cycle latency for result writeback

### 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



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	00101bbb00101111FBBBCCCCC000101
ABSS<.f>	b,limm	00101bbb00101111FBBB111110000101
ABSS<.f>	0,c	0010111000101111F111CCCCC000101
ABSS<.f>	0,limm	0010111000101111F111111110000101
ABSS<.f>	b,u6	00101bbb01101111FBBAuuuuuu000101
ABSS<.f>	0,u6	0010111001101111F111uuuuuu000101

## 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:

```
ADD.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 = SAT16(ABS(c.h0));
```

### Instruction Format

op b, c

### Timing Characteristic

Issue one instruction per cycle; two cycle latency for result writeback

### 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



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	00101bbb00101111FBBBCCCCCC000100
ABSSH<.f>	b,limm	00101bbb00101111FBBB111110000100
ABSSH<.f>	0,c	0010111000101111F111CCCCCC000100
ABSSH<.f>	0,limm	0010111000101111F111111110000100
ABSSH<.f>	b,u6	00101bbb01101111FBBBuuuuuu000100
ABSSH<.f>	0,u6	0010111001101111F111uuuuuu000100

## 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:

```
ADD.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	00101bbb00100110FBBBCCCCCAAAAAA
ADCS<.f>	a,b,u6	00101bbb01100110FBBBuuuuuuAAAAAA
ADCS<.f>	b,b,s12	00101bbb10100110FBBBssssssSSSSSS
ADCS<.cc><.f>	b,b,c	00101bbb11100110FBBBCCCCC0QQQQQ
ADCS<.cc><.f>	b,b,u6	00101bbb11100110FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb00000110FBBBCCCCCCCCAAAAAA
ADDS<.f>	a,b,u6	00101bbb01000110FBBBuuuuuuuuAAAAAA
ADDS<.f>	b,b,s12	00101bbb10000110FBBBssssssSSSSSS
ADDS<.cc><.f>	b,b,c	00101bbb11000110FBBBCCCCCCC0QQQQQ
ADDS<.cc><.f>	b,b,u6	00101bbb11000110FBBBuuuuuuuu1QQQQQ

## 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:

```
ADD.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: accllo = accllo << c.b0;
2: acchi = acchi << c.b0;
3: acchi = acchi << c.b2; accllo = accllo << c.b0;
}
```

### Instruction Format

op c

### Timing Characteristics

Issue one instruction per cycle; three cycle latency for implicit accumulator update

### 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	00101000001011110000CCCCCC111111
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

$a = c \geq 0 ? \text{SAT32}(b << c) : b >> -c$

### Instruction Format

op a, b, c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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) {                                     /* ASLS */
    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	00101bbb00001010FBBBBCCCCCAAAAAAA
ASLS<.f>	a,b,u6	00101bbb01001010FBBBuuuuuuAAAAAAA
ASLS<.f>	b,b,s12	00101bbb10001010FBBBssssssSSSSSS
ASLS<.f><.cc>	b,b,c	00101bbb11001010FBBBCCCCCC0QQQQQ
ASLS<.f><.cc>	b,b,u6	00101bbb11001010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; three cycle latency for implicit accumulator update

### 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	00101001001011110000CCCCCC111111
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 >=0 ? b>>c : SAT32(b<<-c) ;
```

### Instruction Format

op a, b, c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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) {                                     /* ASRS */
    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	00101bbb00001011FBBBCCCCCCAAAAAA
ASRS<.f>	a,b,u6	00101bbb01001011FBBBuuuuuuAAAAAA
ASRS<.f>	b,b,s12	00101bbb10001011FBBBssssssSSSSSS
ASRS<.cc><.f>	b,b,c	00101bbb11001011FBBBCCCCCC0QQQQQ
ASRS<.cc><.f>	b,b,u6	00101bbb11001011FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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 >=0 ? RND32(b>>c) : SAT32(b<<-c) ;
```

### Instruction Format

op a, b, c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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) {                                     /* ASRSR */
    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	00101bbb00001100FBBBCCCCCCCCAAAAAA
ASRSR<.f>	a,b,u6	00101bbb01001100FBBBuuuuuuAAAAAA
ASRSR<.f>	b,b,s12	00101bbb10001100FBBBssssssSSSSSS
ASRSR<.cc><.f>	b,b,c	00101bbb11001100FBBBCCCCCCCC0QQQQQ
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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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:  $A' = A + T \cdot B$ .

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 AUX 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                                     CBFLYHF0R
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) {
    sl = sat(round(ACC0_HI, 16) >> 16, 15, &a.h1);
    sh = sat(round(ACC0_LO, 16) >> 16, 15, &a.h0);
} else {
    sl = 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 || 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
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. Hence 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 example below illustrates how these two instructions can be used

```

cbflyhf1r(*ain++);
for (i = 0; i < max; i++)
{
    *aout++ = cbflyhf0r(*bin++, *twiddle++);
    *bout++ = cbflyhf1r(*ain++);
}

```

## Syntax and Encoding

### Instruction Code

CBFLYHF0R	a,b,c	00110bbb000110111BBBCCCCC <del>AAAAAA</del>
CBFLYHF0R	a,b,u6	00110bbb010110111BBBuuuuuu <del>AAAAAA</del>
CBFLYHF0R	b,b,s12	00110bbb100110111BBBssssss <del>SSSSSS</del>
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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 AUX 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 lpnd
    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
lpnd:

```

## 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. Hence 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 example below illustrates how these two instructions can be used

```

cbflyhf1r(*ain++);
for (i = 0; i < max; i++)
{
    *aout++ = cbflyhf0r(*bin++,
    *twiddle++);
    *bout++ = cbflyhf1r(*ain++);
}
// first A input, ignore first
// return value
// B & twiddle input, A output
// B output, next A input

```

## Syntax and Encoding

### Instruction Code

CBFLYHF1R	b,c	00110bbb001011110BBBCCCCC011001
CBFLYHF1R	b,limm	00110bbb001011110BBB111110011001
CBFLYHF1R	0,c	00110110001011110111CCCCCC011001
CBFLYHF1R	0,limm	00110110001011110111111110011001
CBFLYHF1R	b,u6	00110bbb011011110BBBuuuuuu011001
CBFLYHF1R	0,u6	00110110011011110111uuuuuu011001

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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;                                     /*CMACCHFR*/
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	00110bbb000010011BBBCCCCC <del>AAAAAA</del>
CMACCHFR	a,b,u6	00110bbb010010011BBBuuuuuu <del>AAAAAA</del>
CMACCHFR	b,b,s12	00110bbb100010011BBBssssss <del>SSSSSS</del>
CMACCHFR<.cc>	b,b,c	00110bbb110010011BBBCCCCC <del>0QQQQQ</del>
CMACCHFR<.cc>	b,b,u6	00110bbb110010011BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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                                /*CMACCHNFR
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	00110bbb000010001BBBCCCCC <del>AAAAAA</del>
CMACCHNFR	a,b,u6	00110bbb010010001BBBuuuuuu <del>AAAAAA</del>
CMACCHNFR	b,b,s12	00110bbb100010001BBBssssss <del>SSSSSS</del>
CMACCHNFR<.cc>	b,b,c	00110bbb110010001BBBCCCCC <del>0QQQQQ</del>
CMACCHNFR<.cc>	b,b,u6	00110bbb110010001BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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;                                              CMACHFR
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	00110bbb000001111BBBCCCCC <del>AAAAAA</del>
CMACHFR	a,b,u6	00110bbb010001111BBBuuuuuu <del>AAAAAA</del>
CMACHFR	b,b,s12	00110bbb100001111BBBsaaaaa <del>SSSSSS</del>
CMACHFR<.cc>	b,b,c	00110bbb110001111BBBCCCCC <del>0QQQQQ</del>
CMACHFR<.cc>	b,b,u6	00110bbb110001111BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 CMACHNFR
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	00110bbb000001101BBBCCCCC <del>AAAAAA</del>
CMACHNFR	a,b,u6	00110bbb010001101BBBuuuuuu <del>AAAAAA</del>
CMACHNFR	b,b,s12	00110bbb100001101BBBsssss <del>SSSSSS</del>
CMACHNFR<.cc>	b,b,c	00110bbb110001101BBBCCCCC <del>0QQQQQ</del>
CMACHNFR<.cc>	b,b,u6	00110bbb110001101BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
ACCO_GLO.GUARD = (acclo >> 32) & 0xff;
ACCO_GLO.G      = ((acclo >> 31) & 1) ? ACCO_GLO.GUARD != -1 :
ACCO_GLO.GUARD != 0;
ACCO_LO          = acclo & 0xffff_ffff;
S                |= sat(round(acclo, 16), 31, &alo);
acchi = acc_add(0, addi0+addi1, 40, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
ACCO_GHI.GUARD = (acchi >> 32) & 0xff;
ACCO_GHI.G      = ((acchi >> 31) & 1) ? ACCO_GHI.GUARD != -1 :
ACCO_GHI.GUARD != 0;
ACCO_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, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
ACCO_GLO.G      = 0;
ACCO_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	00110bbb000001011BBBCCCCC <del>AAAAAA</del>
CMPYCHFR	a,b,u6	00110bbb010001011BBBuuuuuu <del>AAAAAA</del>
CMPYCHFR	b,b,s12	00110bbb100001011BBBsaaaaa <del>SSSSSS</del>
CMPYCHFR<.cc>	b,b,c	00110bbb110001011BBBCCCCC <del>0QQQQQ</del>
CMPYCHFR<.cc>	b,b,u6	00110bbb110001011BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
    ACCO_GLO.GUARD = (acclo >> 32) & 0xff;
    ACCO_GLO.G      = ((acclo >> 31) & 1) ? ACCO_GLO.GUARD != -1 :
ACCO_GLO.GUARD != 0;
    ACCO_LO          = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 40, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
    ACCO_GHI.GUARD = (acchi >> 32) & 0xff;
    ACCO_GHI.G      = ((acchi >> 31) & 1) ? ACCO_GHI.GUARD != -1 :
ACCO_GHI.GUARD != 0;
    ACCO_HI          = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addr0+addr1, 32, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
    ACCO_GLO.G      = 0;
    ACCO_LO          = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
    ACCO_GHI.G      = 0;
    ACCO_HI          = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= s | ACCO_GHI.V | ACCO_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

			<b>Instruction Code</b>
CMPYCHNFR	a, b, c		00110bbb000000101BBBCCCCC <del>AAAAAA</del>
CMPYCHNFR	a, b, u6		00110bbb010000101BBBuuuuuu <del>AAAAAA</del>
CMPYCHNFR	b, b, s12		00110bbb100000101BBBssssss <del>SSSSSS</del>
CMPYCHNFR<.cc>	b, b, c		00110bbb110000101BBBCCCCC <del>0QQQQQ</del>
CMPYCHNFR<.cc>	b, b, u6		00110bbb110000101BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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                               CMPYHFR
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, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
        ACCO_GLO.GUARD = (acclo >> 32) & 0xff;
        ACCO_GLO.G      = ((acclo >> 31) & 1) ? ACCO_GLO.GUARD != -1 :
ACCO_GLO.GUARD != 0;
        ACCO_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 40, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
        ACCO_GHI.GUARD = (acchi >> 32) & 0xff;
        ACCO_GHI.G      = ((acchi >> 31) & 1) ? ACCO_GHI.GUARD != -1 :
ACCO_GHI.GUARD != 0;
        ACCO_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(0, addr0+addr1, 32, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_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, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_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, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACCO_GLO.GUARD != 0;
    ACC0_LO        = acclo & 0xffff_ffff;
    s              |= sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 40, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACCO_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     = 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	00110bbb00000011BBBCCCCC <del>AAAAAA</del>
CMPYHFR	a,b,u6	00110bbb010000011BBBuuuuuu <del>AAAAAA</del>
CMPYHFR	b,b,s12	00110bbb100000011BBBssssss <del>SSSSSS</del>
CMPYHFR<.cc>	b,b,c	00110bbb1100000011BBBCCCCC <del>0QQQQQ</del>
CMPYHFR<.cc>	b,b,u6	00110bbb1100000011BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
        ACCO_GLO.GUARD = (acclo >> 32) & 0xff;
        ACCO_GLO.G      = ((acclo >> 31) & 1) ? ACCO_GLO.GUARD != -1 :
ACCO_GLO.GUARD != 0;
        ACCO_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi, 40, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
        ACCO_GHI.GUARD = (acchi >> 32) & 0xff;
        ACCO_GHI.G      = ((acchi >> 31) & 1) ? ACCO_GHI.GUARD != -1 :
ACCO_GHI.GUARD != 0;
        ACCO_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, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
        ACCO_GLO.G      = 0;
        ACCO_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi, 32, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
        ACCO_GHI.G      = 0;
        ACCO_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi, 16), 31, &ahi);

```

```

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	00110bbb000110110BBBCCCCCAAAAAA
CMPYHFMR	a,b,u6	00110bbb010110110BBBuuuuuuAAAAAA
CMPYHFMR	b,b,s12	00110bbb100110110BBBssssssSSSSSS

CMPYHFMR<.cc>	b, b, c	00110bbb110110110BBBBCCCCCC0QQQQQ
CMPYHFMR<.cc>	b, b, u6	00110bbb110110110BBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback; one additional cycle latency and one instruction per two cycles throughput for a 2 cycle complex configuration

### 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                               CMPYHNFR
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, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
    ACCO_GLO.GUARD = (acclo >> 32) & 0xff;
    ACCO_GLO.G      = ((acclo >> 31) & 1) ? ACCO_GLO.GUARD != -1 :
ACCO_GLO.GUARD != 0;
    ACCO_LO          = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 40, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
    ACCO_GHI.GUARD = (acchi >> 32) & 0xff;
    ACCO_GHI.G      = ((acchi >> 31) & 1) ? ACCO_GHI.GUARD != -1 :
ACCO_GHI.GUARD != 0;
    ACCO_HI          = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addr0+addr1, 32, &ACCO_GLO.Z, &ACCO_GLO.N,
&ACCO_GLO.V)
    ACCO_GLO.G      = 0;
    ACCO_LO          = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
    ACCO_GHI.G      = 0;
    ACCO_HI          = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT = s | ACCO_GHI.V | ACCO_GLO.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

## Assembly Code Example

```
CMPYHNFR 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		
CMPYHNFR	a,b,c	00110bbb00000001 BBBCCCCC AAAAAA
CMPYHNFR	a,b,u6	00110bbb010000001 BBBuuuuuu AAAAAA
CMPYHNFR	b,b,s12	00110bbb10000001 BBBssssss SSSSSS
CMPYHNFR<.cc>	b,b,c	00110bbb110000001 BBBCCCCC 0QQQQQ
CMPYHNFR<.cc>	b,b,u6	00110bbb110000001 BBBuuuuuu 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:

```
ADD.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

## Extension Group

(DSP\_DIVSQRT != none )

## Operation

When DSP\_DIVSQRT != none

```
if (cc) {
    acchi = b / c;
    acclo = b % c;
    a = acchi;
}
```

## 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 will happen 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	00101bbb00000100FBBCCCCCAAAAAA
DIV<.f>	a,b,u6	00101bbb01000100FBBCuuuuuuAAAAAA
DIV<.cc><.f>	b,b,c	00101bbb11000100FBBCCCCC0QQQQQ
DIV<.cc><.f>	b,b,u6	00101bbb11000100FBBCuuuuuu1QQQQQ
DIV<.f>	b,b,s12	00101bbb10000100FBBCssssssSSSSSS

## 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:

```
ADD.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

DSP\_DIVSQRT!= no

### 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 abs (b) <= 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	00110bbb00010000FBBBCCCCCCCCAAAAAA
DIVF<.f>	a,b,u6	00110bbb01010000FBBBuuuuuuAAAAAA
DIVF<.f>	b,b,s12	00110bbb10010000FBBBssssssssssss
DIVF<cc><.f>	b,b,u6	00110bbb11010000FBBBuuuuuu1QQQQQ
DIVF<.cc><.f>	b,b,c	0011011011010000F1111111100QQQQQ

## 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:

```
ADD.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

### Extension Group

(dsp\_divsqrt != none)

### Operation

When dsp\_divsqrt != none

```
if (cc) {
    acchi = b / c;
    acclo = b % c;
    a = acchi;
}
```

### 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;
        }
    }
}

```

## Assembly Code Example

```
DIVU r1,r2,r3 ; r1 is assigned r2 / r3
```

## Syntax and Encoding

Instruction Code		
DIVU<.f>	a,b,c	00101bbb00000101FBBBCCCCC <del>AAAAAA</del>
DIVU<.f>	a,b,u6	00101bbb01000101FBBBuuuuuu <del>AAAAAA</del>
DIVU<.cc><.f>	b,b,c	00101bbb11000101FBBBCCCCC <del>0QQQQQ</del>
DIVU<.cc><.f>	b,b,u6	00101bbb11000101FBBBuuuuuu <del>1QQQQQ</del>
DIVU<.f>	b,b,s12	00101bbb10000101FBBBssssss <del>SSSSSS</del>

## 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:

```
ADD.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

DSP dual 16x16 MAC

### 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

### Timing Characteristic

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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



For detailed information about vector operands, see [“Vector Operands” on page 80](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions” on page 82](#) and [“Expansion of Literals” on page 82](#).

### Instruction Code

DMACH<.f>	a,b,c	00101bbb00010010FBBBCCCCCCCCAAAAAA
DMACH<.f>	a,b,u6	00101bbb01010010FBBBuuuuuuAAAAAA
DMACH<.f>	b,b,s12	00101bbb10010010FBBBssssssSSSSSS
DMACH<.cc><.f>	b,b,c	00101bbb11010010FBBBCCCCCCCC0QQQQQ
DMACH<.cc><.f>	b,b,u6	00101bbb11010010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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		
DMACHEBL<.f>	a,b,c	00110bbb00011000FBBBCCCCCCCCAAAAAA
DMACHEBL<.f>	a,b,u6	00110bbb01011000FBBBuuuuuuuAAAAAA
DMACHEBL<.f>	b,b,s12	00110bbb10011000FBBBssssssSSSSSS
DMACHEBL<.cc><.f>	b,b,c	00110bbb11011000FBBBCCCCCCCC0QQQQQ
DMACHEBL<.cc><.f>	b,b,u6	00110bbb11011000FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00011001FBBBCCCCCCCCAAAAAA
DMACHBM<.f>	a,b,u6	00110bbb01011001FBBBuuuuuuAAAAAA
DMACHBM<.f>	b,b,s12	00110bbb10011001FBBBssssssSSSSSS
DMACHBM<.cc><.f>	b,b,c	00110bbb11011001FBBBCCCCCCCC0QQQQQ
DMACHBM<.cc><.f>	b,b,u6	00110bbb11011001FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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 = 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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
s = 0;                                     /* DMACHF
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, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
    ACCO_GHI.GUARD = (acchi >> 32) & 0xff;
    ACCO_GHI.G     = ((acchi >> 31) & 1) ? ACCO_GHI.GUARD != -1 : ACCO_GHI.GUARD != 0;
    ACCO_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, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
    ACCO_GHI.G     = 0;
    ACCO_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, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
        ACCO_GHI.GUARD = (acchi >> 32) & 0xff;
        ACCO_GHI.G     = ((acchi >> 31) & 1) ? ACCO_GHI.GUARD != -1 && ACCO_GHI.GUARD != 0;
        ACCO_HI        = acchi & 0xffff_ffff;
        v             = sat(acchi<<1, 31, &ahi);
    else
        // guard bits are disabled
        acchi      = acc_add(acchi, addhi+addlo, 32, &ACCO_GHI.Z, &ACCO_GHI.N,
&ACCO_GHI.V)
        ACCO_GHI.G     = 0;
        ACCO_HI        = acchi & 0xffff_ffff;
        v             = sat(acchi<<1, 31, &ahi);
    DSP_CTRL.SAT |= s || v || ACCO_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	00101bbb00101100FBBBCCCCCCC <del>AAAAAA</del>
DMACHF<.f>	a, b, u6	00101bbb01101100FBBBuuuuuu <del>AAAAAA</del>
DMACHF<.f>	b, b, s12	00101bbb10101100FBBBssssss <del>SSSSSS</del>
DMACHF<.cc><.f>	b, b, c	00101bbb11101100FBBBCCCCC <del>0QQQQQ</del>
DMACHF<.cc><.f>	b, b, u6	00101bbb11101100FBBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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 = SAT16(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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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;                                     /* DMACHFR */
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	00101bbb00101101FBBBCCCCCCCCAAAAAA
DMACHFR<.f>	a,b,u6	00101bbb01101101FBBBuuuuuuuAAAAAA
DMACHFR<.f>	b,b,s12	00101bbb10101101FBBBssssssSSSSSS
DMACHFR<.cc><.f>	b,b,c	00101bbb11101101FBBBCCCCCCCC0QQQQQ
DMACHFR<.cc><.f>	b,b,u6	00101bbb11101101FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

DSP dual 16x16 MAC

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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_L0 = 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



For detailed information about vector operands, see “[Vector Operands](#)” on page 80. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page 82 and “[Expansion of Literals](#)” on page 82.

### Instruction Code

DMACHU<.f>	a,b,c	00101bbb00010011FBBBCCCCCCCCAAAAAA
DMACHU<.f>	a,b,u6	00101bbb01010011FBBBuuuuuuAAAAAA
DMACHU<.f>	b,b,s12	00101bbb10010011FBBBssssssSSSSSS
DMACHU<.cc><.f>	b,b,c	00101bbb11010011FBBBCCCCCCCC0QQQQQ
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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

---

```
ADD.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](#).

### 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:

```
ADD.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

DSP dual 16x16 MAC

### 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. wo (lower) represents the 32-bit element of an operand.

---

## Instruction Format

op a, b, c

## Syntax Example

DMPYH <.f> a,b,c

## Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

## 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](#)” on page 80. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page 82 and “[Expansion of Literals](#)” on page 82.

Instruction Code		
DMPYH<.f>	a, b, c	00101bbb00010000FBBBCCCCCCCCAAAAAA
DMPYH<.f>	a, b, u6	00101bbb01010000FBBBuuuuuuuAAAAAA
DMPYH<.f>	b, b, s12	00101bbb10010000FBBBssssssSSSSSS
DMPYH<.cc><.f>	b, b, c	00101bbb11010000FBBBCCCCCC0QQQQQ
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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00010110FBBBCCCCCAAAAAA
DMPYHBL<.f>	a,b,u6	00110bbb01010110FBBBuuuuuuAAAAAA
DMPYHBL<.f>	b,b,s12	00110bbb10010110FBBBssssssSSSSSS
DMPYHBL<.cc><.f>	b,b,c	00110bbb11010110FBBBCCCCC0QQQQQ
DMPYHBL<.cc><.f>	b,b,u6	00110bbb11010110FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00010111FBBBCCCCCAAAAAA
DMPYHBM<.f>	a,b,u6	00110bbb01010111FBBBuuuuuuAAAAAA
DMPYHBM<.f>	b,b,s12	00110bbb10010111FBBBssssssSSSSSS
DMPYHBM<.cc><.f>	b,b,c	00110bbb11010111FBBBCCCCC0QQQQQ
DMPYHBM<.cc><.f>	b,b,u6	00110bbb11010111FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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** h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. wo (lower) represents the 32-bit element of an operand.

### Instruction Format

op a, b, c

### Timing Characteristic

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00101bbb00101010FBBBCCCCCCCCAAAAAA
DMPYHF<.f>	a,b,u6	00101bbb01101010FBBBuuuuuuuAAAAAA
DMPYHF<.f>	b,b,s12	00101bbb10101010FBBBssssssSSSSSS
DMPYHF<.cc><.f>	b,b,c	00101bbb11101010FBBBCCCCCCC0QQQQQ
DMPYHF<.cc><.f>	b,b,u6	00101bbb11101010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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           /* DMPYHFR
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	00101bbb00101011FBBBCCCCCAAAAAA
DMPYHFR<.f>	a,b,u6	00101bbb01101011FBBBuuuuuuAAAAAA
DMPYHFR<.f>	b,b,s12	00101bbb10101011FBBBssssssSSSSSS
DMPYHFR<.cc><.f>	b,b,c	00101bbb11101011FBBBCCCCC0QQQQQ
DMPYHFR<.cc><.f>	b,b,u6	00101bbb11101011FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

DSP dual 16x16 MAC

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

### 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:

```
ADD.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](#).

## 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:

```
ADD.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. wo (lower) represents the 32-bit element of an operand.

### Instruction Format

op a, b, c

### Syntax Example

DMPYHWF <.f> a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00101bbb00101000FBBBCCCCAAAAAA
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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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

## Timing Characteristics

Issue one instruction per cycle; three cycle latency for implicit accumulator update

## 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	0010110000101111000CCCCCC111111
FLAGACC	u6	0010110001101111000uuuuuu111111

## GETACC

### Function

Get the accumulator value

### Extension Group

DSP Accumulator

### Operation

```
b = shift_round_sat(acc, c);
```

### Instruction Format

op b,c

### Timing Characteristics

Issue one instruction per cycle; three cycle latency for implicit accumulator update

### 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 will occur 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 remaining MSBs, LSBS will be 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           /* GETACC */
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 &= 0xfffff00_00000000; losat &= 0xfffff00_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	00101bbb001011110BBBCCCCC011000	
GETACC	b, limm	00101bbb001011110BBB111110011000	
GETACC	0, c	00101110001011110111CCCCCC011000	
GETACC	0, limm	00101110001011110111111110011000	

GETACC	b , u6	00101bbb011011110BBBuuuuuu011000
GETACC	0 , u6	00101110011011110111uuuuuu011000

## MAC

### Function

Signed 32x32 multiplication and accumulation.

### Extension Group

DSP 32x32 MAC

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00101bbb00001110FBBBCCCCCAAAAAA
MAC<.f>	a,b,u6	00101bbb01001110FBBBuuuuuuAAAAAA
MAC<.f>	b,b,s12	00101bbb10001110FBBBssssssSSSSSS

MAC<.cc><.f>	b, b, c	00101bbb11001110FBBBBCCCCC0QQQQQ
MAC<.cc><.f>	b, b, u6	00101bbb11001110FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

DSP 32x32 MAC

### 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 writeback.

## 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	00101bbb00011010FBBBCCCCCCCCAAAAAA
MACD<.f>	a,b,u6	00101bbb01011010FBBBuuuuuuuAAAAAA
MACD<.f>	b,b,s12	00101bbb10011010FBBBssssssSSSSSS
MACD<.cc><.f>	b,b,c	00101bbb11011010FBBBCCCCCC0QQQQQ
MACD<.cc><.f>	b,b,u6	00101bbb11011010FBBBuuuuuuu1QQQQQ

## 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:

ADD.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. Return a 64-bit result.

### Extension Group

DSP 32x32 MAC

### 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 writeback.

## 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	00110bbb00010011FBBBCCCCAAAAAA
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:

```
ADD.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

DSP 32x32 MAC

### 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 writeback.

## 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



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

### Instruction Code

MACDU<.f>	a, b, c	00101bbb00011011FBBBCCCCCCCCAAAAAA
MACDU<.f>	a, b, u6	00101bbb01011011FBBBuuuuuuAAAAAA
MACDU<.f>	b, b, s12	00101bbb10011011FBBBssssssSSSSSS
MACDU<.cc><.f>	b, b, c	00101bbb11011011FBBBCCCCCCC0QQQQQ
MACDU<.cc><.f>	b, b, u6	00101bbb11011011FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00001100FBBBCCCCCCCCAAAAAA
MACF<.f>	a, b, u6	00110bbb01001100FBBBuuuuuuuAAAAAA
MACF<.f>	b, b, s12	00110bbb10001100FBBBssssssSSSSSS
MACF<.cc><.f>	b, b, c	00110bbb11001100FBBBCCCCCC0QQQQQ
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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### STATUS32 Flags Affected

Z	<input checked="" type="checkbox"/>	= Set if the result is zero
N	<input checked="" type="checkbox"/>	= Set to the resulting sign is negative
C	<input type="checkbox"/>	= Unchanged
V	<input checked="" type="checkbox"/>	= 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 <del>AAAAAA</del>
MACFR<.f>	a, b, u6	00110bbb01001101FBBB <u>uuuuuuAAAAAA</u>
MACFR<.f>	b, b, s12	00110bbb10001101FBBB <del>s</del> ssssssSSSSSS
MACFR<.cc><.f>	b, b, c	00110bbb11001101FBBBCCCCC <del>0</del> QQQQQ
MACFR<.cc><.f>	b, b, u6	00110bbb11001101FBBB <u>uuuuuu1QQQQQ</u>

## 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:

```
ADD.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

DSP 32x32 MAC

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; two cycle latency for explicit writeback

### 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	00101bbb00001111FBBBCCCCCCCCAAAAAA
MACU<.f>	a,b,u6	00101bbb01001111FBBBuuuuuuuuAAAAAA
MACU<.f>	b,b,s12	00101bbb10001111FBBBssssssSSSSSS
MACU<.cc><.f>	b,b,c	00101bbb11001111FBBBCCCCCCCC0QQQQQ
MACU<.cc><.f>	b,b,u6	00101bbb11001111FBBBuuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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                                /* MACWHFM
s = 0;                                                               */
if (DSP_CTRL.PA)                                                 if (DSP_CTRL.GE)
    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

```

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	00110bbb00100010FBBBCCCCCCCCAAAAAA
MACWHFM<.f>	a, b, u6	00110bbb01100010FBBBuuuuuuAAAAAA
MACWHFM<.f>	b, b, s12	00110bbb10100010FBBBssssssSSSSSS
MACWHFM<.cc><.f>	b, b, c	00110bbb11100010FBBBCCCCCCC0QQQQQ
MACWHFM<.cc><.f>	b, b, u6	00110bbb11100010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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                                /* MACWHFMR
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

```

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	00110bbb00100011FBBBCCCCCCCCAAAAAA
MACWHFMR<.f>	a, b, u6	00110bbb01100011FBBBuuuuuuAAAAAA
MACWHFMR<.f>	b, b, s12	00110bbb10100011FBBBssssssSSSSSS
MACWHFMR<.cc><.f>	b, b, c	00110bbb11100011FBBBCCCCCC0QQQQQ
MACWHFMR<.cc><.f>	b, b, u6	00110bbb11100011FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; two cycle latency for explicit writeback

### 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                                /* MACWHL
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_ff;                                         */
ACC0_LO = accwide & 0xffff_ff;                                              */
a = accwide & 0xffff_ff;                                                 */
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	00110bbb00011101FBBBCCCCCCCCAAAAAA
MACWHL<.f>	a,b,u6	00110bbb01011101FBBBuuuuuuuAAAAAA
MACWHL<.f>	b,b,s12	00110bbb10011101FBBBssssssssSSSSSS
MACWHL<.cc><.f>	b,b,c	00110bbb11011101FBBBCCCCCCCC0QQQQQ
MACWHL<.cc><.f>	b,b,u6	00110bbb11011101FBBBuuuuuuu1QQQQQ

## 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:

ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00100110FBBBCCCCCCCCAAAAAA
MACWHFL<.f>	a, b, u6	00110bbb01100110FBBBuuuuuuAAAAAA
MACWHFL<.f>	b, b, s12	00110bbb10100110FBBBssssssSSSSSS
MACWHFL<.cc><.f>	b, b, c	00110bbb11100110FBBBCCCCCCC0QQQQQ
MACWHFL<.cc><.f>	b, b, u6	00110bbb11100110FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00100111FBBBCCCCCAAAAAA
MACWHFLR<.f>	a, b, u6	00110bbb01100111FBBBuuuuuuAAAAAA
MACWHFLR<.f>	b, b, s12	00110bbb10100111FBBBssssssSSSSSS
MACWHFLR<.cc><.f>	b, b, c	00110bbb11100111FBBBCCCCC0QQQQQ
MACWHFLR<.cc><.f>	b, b, u6	00110bbb11100111FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two-cycle latency for implicit accumulator update; two cycle latency for explicit writeback.

### 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	00110bbb00101000FBBBCCCCCCCCAAAAAA
MACWHKL<.f>	a,b,u6	00110bbb01101000FBBBuuuuuuAAAAAA
MACWHKL<.f>	b,b,s12	00110bbb10101000FBBBssssssSSSSSS
MACWHKL<.cc><.f>	b,b,c	00110bbb11101000FBBBCCCCCCC0QQQQQ
MACWHKL<.cc><.f>	b,b,u6	00110bbb11101000FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; two cycle latency for explicit writeback

### 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 = (accwide >> 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	00110bbb00101001FBBBCCCCCCCCAAAAAA
MACWHKUL<.f>	a,b,u6	00110bbb01101001FBBBuuuuuuuAAAAAA
MACWHKUL<.f>	b,b,s12	00110bbb10101001FBBBssssssSSSSSS
MACWHKUL<.cc><.f>	b,b,c	00110bbb11101001FBBBCCCCCCCC0QQQQQ
MACWHKUL<.cc><.f>	b,b,u6	00110bbb11101001FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; two cycle latency for explicit writeback

### 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          /* MACWHUL
// 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;
}

```

## 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	00110bbb00011111FBBBCCCCCCCCAAAAAA
MACWHUL<.f>	a,b,u6	00110bbb01011111FBBBuuuuuuAAAAAA
MACWHUL<.f>	b,b,s12	00110bbb10011111FBBBssssssSSSSSS
MACWHUL<.cc><.f>	b,b,c	00110bbb11011111FBBBCCCCCCCC0QQQQQ
MACWHUL<.cc><.f>	b,b,u6	00110bbb11011111FBBBuuuuuu1QQQQQ

## 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:

ADD.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\_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  $2^{VW} <$  container size, the associated address pointer is not updated. Where, VW is the AGU\_AUX\_MODx [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 = unmrber 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	00101101001011110000CCCCCCCC111111
	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 registers as input. If the operand registers 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  $2^{VW} <$  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);  
}
```

## 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	00101bbb011011110BBBBuuuuuu111110

## MPY

### Function

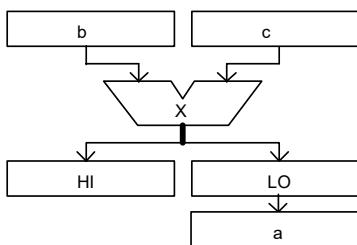
32 x 32 Signed Multiply, returning least-significant 32-bit word of the result.

### Extension Group

- has\_dsp==true – the MPY instruction uses the DSP datapath

### Operation

```
if (cc) a = (signed) (b * c) & 0xFFFF_FFFF;
```



### Instruction Format

op a, b, c

### Syntax Example

MPY<.f> a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for explicit writeback

### 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	00100bbb00011010FBBCCCCCAAAAAA
MPY<.f>	a,b,u6	00100bbb01011010FBBAuuuuuuAAAAAA
MPY<.f>	b,b,s12	00100bbb10011010FBBSssssssSSSSSS
MPY<.cc><.f>	b,b,c	00100bbb11011010FBBCCCCCC0QQQQQ
MPY<.cc><.f>	b,b,u6	00100bbb11011010FBBAuuuuuu1QQQQQ
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:

```
ADD.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.

### Extension Group

has\_dsp==true

### Operation

$t = b * c;$

$b = t.w0;$

### Instruction Format

op b, b, c

### Syntax Example

MPY\_S b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for explicit writeback

### 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

DSP 32x32 MAC

### Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```

### Instruction Format

op A, b, c

### Syntax Example

MPYD <.f> A,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 writeback.

## 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](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

### Instruction Code

MPYD<.f>	a,b,c	00101bbb00011000FBBBCCCCCCCCAAAAAA
MPYD<.f>	a,b,u6	00101bbb01011000FBBBuuuuuuAAAAAA
MPYD<.f>	b,b,s12	00101bbb10011000FBBBssssssSSSSSS
MPYD<.cc><.f>	b,b,c	00101bbb11011000FBBBCCCCCC0QQQQQ
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:

```
ADD.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 writeback.

## 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	00110bbb00010010FBBBCCCCCCCCAAAAAA
MPYDF<.f>	a, b, u6	00110bbb01010010FBBBuuuuuuuAAAAAA
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:

```
ADD.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

DSP 32x32 MAC

### Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```

### Instruction Format

op A, b, c

### Syntax Example

MPYDU <.f> A,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 writeback.

## 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



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

### Instruction Code

MPYDU<.f>	a,b,c	00101bbb00011001FBBBBCCCCC <del>AAAAAA</del>
MPYDU<.f>	a,b,u6	00101bbb01011001FBBBuuuuuu <del>AAAAAA</del>
MPYDU<.f>	b,b,s12	00101bbb10011001FBBBssssss <del>SSSSSS</del>
MPYDU<.cc><.f>	b,b,c	00101bbb11011001FBBBCCCCC <del>0QQQQQ</del>
MPYDU<.cc><.f>	b,b,u6	00101bbb11011001FBBBuuuuuu <del>1QQQQQ</del>

## 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:

ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00001010FBBBCCCCCCCCAAAAAA
MPYF<.f>	a,b,u6	00110bbb01001010FBBBuuuuuuuuAAAAAA

MPYF<.f>	b, b, s12	00110bbb10001010FBBBssssssSSSSSS
MPYF<.cc><.f>	b, b, c	00110bbb110001010FBBBCCCCCC0QQQQQ
MPYF<.cc><.f>	b, b, u6	00110bbb110001010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00001011FBBBCCCCCCCCAAAAAA
MPYFR<.f>	a,b,u6	00110bbb01001011FBBBuuuuuuuuAAAAAA

MPYFR<.f>	b, b, s12	00110bbb10001011FBBBssssssSSSSSS
MPYFR<.cc><.f>	b, b, c	00110bbb110001011FBBBCCCCCC0QQQQQ
MPYFR<.cc><.f>	b, b, u6	00110bbb110001011FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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



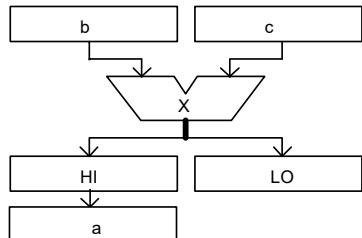
The MPYH mnemonic is deprecated.

### Extension Group

- has\_dsp == true – the MPY instruction uses the DSP datapath

### Operation

```
if (cc) a = (signed) (b * c) >> 32;
```



### Instruction Format

op a, b, c

### Syntax Example

MPYM<.f> a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for explicit writeback

### STATUS32 Flags Affected

Z		= Set when the destination register is zero.
N		= Set if 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	00100bbb00011011FB <del>BB</del> CCCCCCCCAAAAAA
MPYM<.f>	a,b,u6	00100bbb01011011FB <del>BB</del> BBBBBBBBAAAAAA
MPYM<.f>	b,b,s12	00100bbb10011011FB <del>BB</del> ssssssSSSSSS
MPYM<.cc><.f>	b,b,c	00100bbb11011011FB <del>BB</del> CCCCCC0QQQQQ
MPYM<.cc><.f>	b,b,u6	00100bbb11011011FB <del>BB</del> BBBBBBBB1QQQQQ

## 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:

```
ADD.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



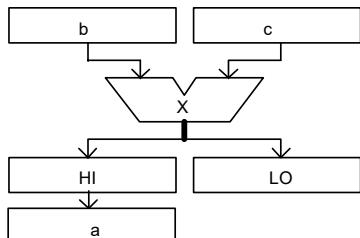
The MPYHU mnemonic is deprecated.

### Extension Group

- has\_dsp == true – the MPY instruction uses the DSP datapath

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for explicit writeback

### 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	00100bbb00011100FBBBCCCCCCCCAAAAAA
MPYMU<.f>	a,b,u6	00100bbb01011100FBBBuuuuuuAAAAAA
MPYMU<.f>	b,b,s12	00100bbb10011100FBBBssssssSSSSSS
MPYMU<.cc><.f>	b,b,c	00100bbb11011100FBBBCCCCCCCC0QQQQQ
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:

```
ADD.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

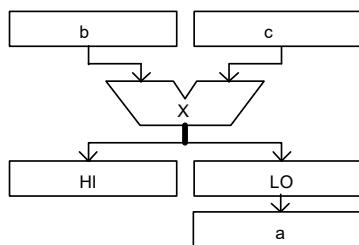
32 x 32 Unsigned Multiply, returning least-significant 32-bit word of result.

### Extension Group

- has\_dsp ==true – the MPY instruction uses the DSP datapath

### Operation

```
if (cc) a = (unsigned) (b * c) & 0xFFFF_FFFF;
```



### Instruction Format

op a, b, c

### Syntax Example

MPYU<.f> a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for explicit writeback

### 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                                /* MPYU */
t = (unsigned)b * c;                                                 /* when has_dsp==true
a = (t & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = 0;
    V_flag = (t >> 32) != 0;
}
```

## 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	00100bbb00011101FBBCCCCCAAAAAA
MPYU<.f>	a,b,u6	00100bbb01011101FBBCuuuuuuAAAAAA
MPYU<.f>	b,b,s12	00100bbb10011101FBBCssssssSSSSSS
MPYU<.cc><.f>	b,b,c	00100bbb11011101FBBCCCCCC0QQQQQ
MPYU<.cc><.f>	b,b,u6	00100bbb11011101FBBCuuuuuu1QQQQQ

## 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:

```
ADD.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

### Extension Group

- has\_dsp == true – the MPY instruction uses the DSP datapath

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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;
}
```

## 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	00100bbb00011111FBBBCCCCCCCCAAAAAA
MPYUW<.f>	a,b,u6	00100bbb01011111FBBBuuuuuuAAAAAA
MPYUW<.f>	b,b,s12	00100bbb10011111FBBBssssssSSSSSS
MPYUW<.cc><.f>	b,b,c	00100bbb11011111FBBBCCCCCCC0QQQQQ
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:

```
ADD.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.

### Extension Group

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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for explicit writeback

### 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

### Extension Group

- has\_dsp == true – the MPY instruction uses the DSP datapath

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for explicit writeback

### 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	00100bbb00011110FBBBCCCCCCCCAAAAAA
MPYW<.f>	a,b,u6	00100bbb01011110FBBBuuuuuuAAAAAA
MPYW<.f>	b,b,s12	00100bbb10011110FBBBssssssSSSSSS
MPYW<.cc><.f>	b,b,c	00100bbb11011110FBBBCCCCCCC0QQQQQ
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:

```
ADD.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.

### Extension Group

DSP 32x32 MAC

### Operation

```
t = b.h0 * c.h0;
b = t.w0;
```

### Instruction Format

op b, c

### Syntax Example

MPYW\_S<.f> b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for explicit writeback

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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                         /* MPYWHFL
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;
}

```

## 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	00110bbb00100100FBBCCCCCAAAAAA
MPYWHFL<.f>	a,b,u6	00110bbb01100100FBBCuuuuuuAAAAAA

MPYWHFL<.f>	b, b, s12	00110bbb10100100FBBBBssssssSSSSSS
MPYWHFL<.cc><.f>	b, b, c	00110bbb11100100FBBBCCCCC0QQQQQ
MPYWHFL<.cc><.f>	b, b, u6	00110bbb11100100FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110 <b>bbb</b> 00100101FBBBCCCCAAAAAA
--------------	-------	---

MPYWHFLR<.f>	a, b, u6	00110bbb01100101FBBBuuuuuuAAAAAA
MPYWHFLR<.f>	b, b, s12	00110bbb10100101FBBBssssssSSSSSS
MPYWHFLR<.cc><.f>	b, b, c	00110bbb11100101FBBBCCCCCC0QQQQQ
MPYWHFLR<.cc><.f>	b, b, u6	00110bbb11100101FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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                                /* MPYWHFM
s = 0;                                                               */
if (DSP_CTRL.PA)                                                 if (DSP_CTRL.GE)
    if (ACC0_GHI.V = sat_en(add, 71, &accwide);                         accwide >> 64) & 0xff;
        ACC0_GHI.GUARD = (accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :      ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;                                              ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);                           v = sat(accwide>>32, 31, &a);
        ACC0_GHI.V = sat(add, 63, &accwide);                               ACC0_GHI.G = 0;
        ACC0_GHI.G = 0;                                                 v = sat(accwide>>32, 31, &a);
    else
        if (DSP_CTRL.GE)                                                 if (DSP_CTRL.GE)
            ACC0_GHI.V = sat_en(b * c.h1 << 16, 71, &accwide);           accwide >> 64) & 0xff;
            ACC0_GHI.GUARD = (accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :      ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;                                              ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h1 << 16, 63, &accwide);             ACC0_GHI.G = 0;
            ACC0_GHI.G = 0;                                               v = sat(accwide>>31, 31, &a);
        v = sat(accwide>>31, 31, &a);                                     DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;                            ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;                          ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;                                       ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;                                         ACC0_GHI.N = accwide < 0;
        if F {                                                       if F {
            Z_flag = a = 0;                                         Z_flag = a = 0;
            N_flag = a < 0;                                         N_flag = a < 0;
            V_flag = v;                                           V_flag = v;
        }
    }
}

```

## 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	00110bbb00100000FBBCCCCCAAAAAA
MPYWHFM<.f>	a,b,u6	00110bbb01100000FBBuuuuuuAAAAAA

MPYWHFM<.f>	b, b, s12	00110bbb10100000FBBBBssssssSSSSSS
MPYWHFM<.cc><.f>	b, b, c	00110bbb11100000FBBBCCCCC0QQQQQ
MPYWHFM<.cc><.f>	b, b, u6	00110bbb11100000FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00100001FBBBCCCCCCCCAAAAAA
MPYWHFMR<.f>	a,b,u6	00110bbb01100001FBBBuuuuuuAAAAAA
MPYWHFMR<.f>	b,b,s12	00110bbb10100001FBBBssssssSSSSSS

MPYWHFMR<.cc><.f>	b, b, c	00110bbb11100001FBBBBCCCCC0QQQQQ
MPYWHFMR<.cc><.f>	b, b, u6	00110bbb11100001FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; two cycle latency for explicit writeback

### 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                                /* MPYWHL
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	00110bbb00011100FBBBBCCCCCAAAAAA
MPYWHL<.f>	a,b,u6	00110bbb01011100FBBBuuuuuuAAAAAA
MPYWHL<.f>	b,b,s12	00110bbb10011100FBBBssssssSSSSSS
MPYWHL<.cc><.f>	b,b,c	00110bbb11011100FBBBCCCCC0QQQQQ
MPYWHL<.cc><.f>	b,b,u6	00110bbb11011100FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; two cycle latency for explicit writeback

### 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                                /* MPYWHUL
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;
}

```

## 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	00110bbb00011110FBBBCCCCCAAAAAA
MPYWHUL<.f>	a,b,u6	00110bbb01011110FBBBuuuuuuAAAAAA
MPYWHUL<.f>	b,b,s12	00110bbb10011110FBBBssssssSSSSSS
MPYWHUL<.cc><.f>	b,b,c	00110bbb11011110FBBBCCCCC0QQQQQ
MPYWHUL<.cc><.f>	b,b,u6	00110bbb11011110FBBBuuuuuu1QQQQQ

## 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:

ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; two cycle latency for explicit writeback

### 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	00110bbb00101010FBBBBCCCCCAAAAAAA
MPYWHKL<.f>	a,b,u6	00110bbb01101010FBBBuuuuuuAAAAAAA
MPYWHKL<.f>	b,b,s12	00110bbb10101010FBBBssssssSSSSSS
MPYWHKL<.cc><.f>	b,b,c	00110bbb11101010FBBBCCCCCC0QQQQQ
MPYWHKL<.cc><.f>	b,b,u6	00110bbb11101010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; two cycle latency for explicit writeback

### 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	<p>; Unsigned integer multiplication of r2 and r3          ; shift result by 16 bits, and return the shifted          ; result in register r0</p>
-------------------	---

## Syntax and Encoding

### Instruction Code

MPYWHKUL<.f>	a,b,c	00110bbb00101011FBBBCCCCCAAAAAA
MPYWHKUL<.f>	a,b,u6	00110bbb01101011FBBBuuuuuuAAAAAA
MPYWHKUL<.f>	b,b,s12	00110bbb10101011FBBBssssssSSSSSS
MPYWHKUL<.cc><.f>	b,b,c	00110bbb11101011FBBBCCCCC0QQQQQ
MPYWHKUL<.cc><.f>	b,b,u6	00110bbb11101011FBBBuuuuuu1QQQQQ

## 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:

ADD.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 writeback.

## 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	00110bbb00010101FBBBCCCCCCCCAAAAAA
MSUBDF<.f>	a, b, u6	00110bbb01010101FBBBuuuuuuAAAAAA
MSUBDF<.f>	b, b, s12	00110bbb10010101FBBBssssssSSSSSS
MSUBDF<.cc><.f>	b, b, c	00110bbb11010101FBBBCCCCCCC0QQQQQ
MSUBDF<.cc><.f>	b, b, u6	00110bbb11010101FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00001110FBBBCCCCCAAAAAA
MSUBF<.f>	a, b, u6	00110bbb01001110FBBBuuuuuuAAAAAA
MSUBF<.f>	b, b, s12	00110bbb10001110FBBBssssssSSSSSS
MSUBF<.cc><.f>	b, b, c	00110bbb11001110FBBBCCCCC0QQQQQ
MSUBF<.cc><.f>	b, b, u6	00110bbb11001110FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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

<b>MSUBFR r1,r2,r3</b>	; 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	00110bbb00001111FBBBCCCCCCCCAAAAAA
MSUBFR<.f>	a, b, u6	00110bbb01001111FBBBuuuuuuAAAAAA
MSUBFR<.f>	b, b, s12	00110bbb10001111FBBBssssssSSSSSS
MSUBFR<.cc><.f>	b, b, c	00110bbb11001111FBBBCCCCCCC0QQQQQ
MSUBFR<.cc><.f>	b, b, u6	00110bbb11001111FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00010100FBBBCCCCCAAAAAA
MSUBWHFL<.f>	a, b, u6	00110bbb01010100FBBBuuuuuuAAAAAA
MSUBWHFL<.f>	b, b, s12	00110bbb10010100FBBBssssssSSSSSS
MSUBWHFL<.cc><.f>	b, b, c	00110bbb11010100FBBBCCCCC0QQQQQ
MSUBWHFL<.cc><.f>	b, b, u6	00110bbb11010100FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00011010FBBBCCCCCCCCAAAAAA
MSUBWHFLR<.f>	a, b, u6	00110bbb01011010FBBBuuuuuuAAAAAA
MSUBWHFLR<.f>	b, b, s12	00110bbb10011010FBBBssssssssssss
MSUBWHFLR<.cc><.f>	b, b, c	00110bbb11011010FBBBCCCCCCCC0QQQQQ
MSUBWHFLR<.cc><.f>	b, b, u6	00110bbb11011010FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00101100FBBBBCCCCC <del>AAAAAA</del>
MSUBWHFM<.f>	a, b, u6	00110bbb01101100FBBBuuuuuu <del>AAAAAA</del>
MSUBWHFM<.f>	b, b, s12	00110bbb10101100FBBBssssss <del>SSSSSS</del>
MSUBWHFM<.cc><.f>	b, b, c	00110bbb11101100FBBBCCCCC <del>0QQQQQ</del>
MSUBWHFM<.cc><.f>	b, b, u6	00110bbb11101100FBBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00101101FBBBBCCCCC <del>AAAAAA</del>
MSUBWHFMR<.f>	a, b, u6	00110bbb01101101FBBBuuuuuu <del>AAAAAA</del>
MSUBWHFMR<.f>	b, b, s12	00110bbb10101101FBBBssssss <del>SSSSSS</del>
MSUBWHFMR<.cc><.f>	b, b, c	00110bbb11101101FBBBCCCCC <del>0QQQQQ</del>
MSUBWHFMR<.cc><.f>	b, b, u6	00110bbb11101101FBBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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 =SAT32 (-c);
```

### Instruction Format

op b,c

### Syntax Example

```
NEGS<.f> b,c
```

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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 ? 0xffff_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	00101bbb00000111FBBBCCCCCC000111
NEGS<.f>	b, limm	00101bbb00101111FBBB111110000111
NEGS<.f>	0, c	0010111000101111F111CCCCCC000111
NEGS<.f>	0, limm	0010111000101111F111111110000111
NEGS<.f>	b, u6	00101bbb01000111FBBAuuuuuu000111
NEGS<.f>	0, u6	0010111001101111F111uuuuuu000111

## 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:

```
ADD.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 =SAT16 (-c) ;

### Instruction Format

op b,c

### Syntax Example

NEGSH<.f> b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb00101111FBBBCCCCCC000110
NEGSH<.f>	b,limm	00101bbb00101111FBBB111110000110
NEGSH<.f>	0,c	0010111000101111F111CCCCCC000110
NEGSH<.f>	0,limm	0010111000101111F111111110000110
NEGSH<.f>	0,u6	0010111001101111F111uuuuuu000110
NEGSH<.f>	b,u6	00101bbb01101111FBBBuuuuuu000110

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; three cycle latency for implicit accumulator update

### 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 <= B <= 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	00101110001011110111CCCCCC011001
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:

```
ADD.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](#).

## QMACH

### Function

Quad 16x16 multiply and accumulate

### Extension Group

MPY\_OPTION == 9

### Operation

```
if (cc) {
    result = acc + (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3);
    A = result;
    acc = result;
}
```



**Note** h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. A, B, and C are register pairs representing 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see “[Data Formats](#)” on page [80](#).

### Instruction Format

op A,B,C

### Syntax Example

QMACH <.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

The B and C operands specify 64-bit values, each containing four 16-bit elements. Each pair of four elements in B and C are multiplied to form four 32-bit products. The four products are added to the accumulator to form the result. The result is then assigned to the destination register pair and the accumulator.

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is set if an overflow occurs during accumulation. This instruction never clears the V flag. The sign (N) flag is set to the resulting sign of the accumulator.

## Pseudo Code

```

if (DSP_CTRL.GE)                                     /*QMACH*/
    accwide_org = sext(ACC0_GHI.GUARD) << 64 | unsigned(ACC0_HI) << 32 |
    unsigned(ACC0_LO)
else
    accwide_org = sext(ACC0_HI << 32) | unsigned(ACC0_LO)
accwide = accwide_org + b.h0 * c.h0 + b.h1 * c.h1 + b.h2 * c.h2 + b.h3 *
c.h3;
if (DSP_CTRL.GE)           // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
a = accwide & 0xffff_ffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    V_flag |= rem != 0 && rem != -1;
    N_flag = rem & 1;
}

```

## Assembly Code Example

```

QMACH r0,r2,r4      ; quad 16x16 multiplication and accumulation of (r3,r2)
                     ; and (r5,r4). The result is stored in (r1,r0)

```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

### Instruction Code

QMACH<.f>	a,b,c	00101bbb00110100FBBBCCCCCCCCAAAAAA
QMACH<.f>	a,b,u6	00101bbb01110100FBBBuuuuuuAAAAAA
QMACH<.f>	b,b,s12	00101bbb10110100FBBBssssssSSSSSS
QMACH<.cc><.f>	b,b,c	00101bbb11110100FBBBCCCCCCCC0QQQQQ
QMACH<.cc><.f>	b,b,u6	00101bbb11110100FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## 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:

```
ADD.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

### Extension Group

(dsp\_divsqrt!=none)

### Operation

When dsp\_divsqrt!=none

```
if (cc) {
    acchi = b / c;
    acclo = b % c;
    a = acclo;
}
```

### 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 B-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;
        }
    }
}

```

## Assembly Code Example

REM r1,r2,r3	; r1 is assigned r2 % r3
--------------	--------------------------

## Syntax and Encoding

### Instruction Code

REM<.f>	a,b,c	00101bbb00001000FBBCCCCCAAAAAA
REM<.cc><.f>	b,b,c	00101bbb11001000FBBCCCCC0QQQQQ
REM<.cc><.f>	b,b,limm	00101bbb11001000FBBC1111100QQQQQ
REM<.cc><.f>	b,b,u6	00101bbb11001000FBBCuuuuuu1QQQQQ
REM<.f>	b,b,s12	00101bbb10001000FBBCssssssSSSSSS

## 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:

```
ADD.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

### Extension Group

(dsp\_divsqrt!=none)

### Operation

When dsp\_divsqrt!=none

```
if (cc) {
    acchi = b / c;
    acclo = b % c;
    a = acclo;
}
```

### 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). 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 B-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;
        }
    }
}

```

## Assembly Code Example

REMU r1,r2,r3	; r1 is assigned r2 % r3
---------------	--------------------------

## Syntax and Encoding

### Instruction Code

REMU<.f>	a,b,c	00101bbb00001001FBBBCCCCCAAAAAA
REMU<.f>	a,b,u6	00101bbb01001001FBBBuuuuuuAAAAAA
REMU<.cc><.f>	b,b,c	00101bbb11001001FBBBCCCCC0QQQQQ

**Instruction Code**

REMU<.cc><.f>	b, b, u6	00101bbb110001001FBBBuuuuuu1QQQQQ
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:

```
ADD.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 = SAT16(RND16(c));
```

### Instruction Format

op b,c

### Syntax Example

RNDH<.f> b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb00101111FB <del>BBB</del> CCCCCC000011
RNDH<.f>	b,limm	00101bbb00101111FB <del>BBB</del> 111110000011
RNDH<.f>	0,c	0010111000101111F111CCCCC000011
RNDH<.f>	0,limm	0010111000101111F11111111000011
RNDH<.f>	b,u6	00101bbb01101111FB <del>BBB</del> uuuuuu000011
RNDH<.f>	0,u6	0010111001101111F111uuuuuu000011

## 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:

```
ADD.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 =SAT16(c);
```

### Instruction Format

op b,c

### Syntax Example

SATH<.f> b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb00101111FBBBCCCCCC000010
SATH<.f>	b,limm	00101bbb00101111FBBB111110000010
SATH<.f>	0,c	0010111000101111F111CCCCCC000010
SATH<.f>	0,limm	0010111000101111F111111110000010
SATH<.f>	b,u6	00101bbb01101111FBuuuuuu000010
SATH<.f>	0,u6	0010111001101111F111uuuuuu000010

## 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:

```
ADD.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
```

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb00101111FBBBCCCCCC011010
SATF<.f>	b, u6	00101bbb01101111FBBBuuuuuu011010

## 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:

```
ADD.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	00101bbb00100111FBBBCCCCCCCCAAAAAA
SBCS<.f>	a,b,u6	00101bbb01100111FBBBuuuuuuuuAAAAAA
SBCS<.f>	b,b,s12	00101bbb10100111FBBBssssssssSSSSSS
SBCS<.cc><.f>	b,b,c	00101bbb11100111FBBBCCCCCCCC0QQQQQ
SBCS<.cc><.f>	b,b,u6	00101bbb11100111FBBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update

### 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 will be 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) {                                     /* SETACC */
    // 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	00101bbb000011011BBBCCCCCAAAAAA
SETACC	a, b, u6	00101bbb010011011BBBuuuuuuAAAAAA
SETACC	b, b, s12	00101bbb100011011BBBssssssSSSSSS
SETACC<.cc>	b, b, c	00101bbb110011011BBBCCCCC0QQQQQ
SETACC<.cc>	b, b, u6	00101bbb110011011BBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per 16 cycles; 17 cycle latency for implicit accumulator update; 18 cycle latency for explicit writeback

### 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) {                                     /* SQRT */
    ACC0_HI = sqrt(c);
    ACC0_LO = c - ACC0_HI*ACC0_HI;
    b = ACC0_HI;
    if (F == 1) {
        Z_flag = (b == 0) ? 1 : 0;
    }
}

```

## 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	00101bbb011011110BBBuuuuuuu110000

## 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:

```
ADD.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, yielding a Q31 fractional result.

### 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

### Timing Characteristics

Issue one instruction per 32 cycles; 33 cycle latency for implicit accumulator update; 34 cycle latency for explicit writeback

### STATUS32 Flags Affected

Z		= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

Compute the square-root of an unsigned fractional 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb00000111FBBBCCCCCCCCAAAAAA
SUBS<.f>	a,b,u6	00101bbb01000111FBBBuuuuuuuuuAAAAAA
SUBS<.f>	b,b,s12	00101bbb10000111FBBBssssssssSSSSSS
SUBS<.cc><.f>	b,b,c	00101bbb110000111FBBBCCCCCCCC0QQQQQ
SUBS<.cc><.f>	b,b,u6	00101bbb110000111FBBBuuuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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);
h = ((c>>16) & 0x0000_ffff);
l = ((l & 0x8000) != 0) ? -l : l;
h = ((h & 0x8000) != 0) ? -h : h;
b = (h << 16) | (l & 0x0000_ffff);
/* VABS2H */

```

## 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	00101bbb00101110BBBCCCCC101000
VABS2H	b,limm	00101bbb00101110BBB111110101000
VABS2H	0,c	0010111000101110111CCCCC101000
VABS2H	0,limm	0010111000101110111111110101000
VABS2H	b,u6	00101bbb01101110BBBuuuuuu101000
VABS2H	0,u6	0010111001101110111uuuuuu101000

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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);
h = ((c>>16) & 0x0000_ffff);
s = (l == 0x8000) || (h == 0x8000);
l = l == 0x8000 ? 0x7fff : (((l & 0x8000) != 0) ? -l : l);
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	00101bbb001011110BBBCCCCC101001
VABSS2H	b,limm	00101bbb001011110BBB111110101001
VABSS2H	0,c	00101110001011110111CCCCCC101001
VABSS2H	0,limm	00101110001011110111111110101001
VABSS2H	b,u6	00101bbb011011110BBBuuuuuu101001
VABSS2H	0,u6	00101110011011110111uuuuuu101001

## 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:

```
ADD.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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VADD2H a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

bl = b & 0x0000_ffff;
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
al = bl + cl;
ah = bh + ch;
a = (al & 0x0000_ffff) | (ah << 16);

```

## Assembly Code Example

```
VADD2H r1,r2,r3 ; dual SIMD 16-bit addition of r2 and r3
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page 80. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page 82 and “[Expansion of Literals](#)” on page 82.

### Instruction Code

VADD2H	a,b,c	00101bbb000101000 BBBCCCCC AAAAAAA
VADD2H	a,b,u6	00101bbb010101000 BBBuuuuuu AAAAAAA
VADD2H	b,b,s12	00101bbb100101000 BBBssssss SSSSSS
VADD2H<.cc>	b,b,c	00101bbb110101000 BBBCCCCC 0QQQQQ
VADD2H<.cc>	b,b,u6	00101bbb110101000 BBBuuuuuu 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:

```
ADD.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

```
a.b3 = b.b3+c.b3;
a.b2 = b.b2+c.b2;
a.b1 = b.b1+c.b1;
a.b0 = b.b0+c.b0;
```



**Note** h0 and h1 represent the 16-bit elements of an operand.

### Instruction Format

op a,b,c

### Syntax Example

VADD4B a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

bll = b & 0x0000_00ff;                                /* VADD4B */
blh = b & 0x0000_ff00;
bhl = b & 0x00ff_0000;
bhh = b & 0xff00_0000;
c1l = c & 0x0000_00ff;
clh = c & 0x0000_ff00;
chl = c & 0x00ff_0000;
chh = c & 0xff00_0000;
all = bll + c1l;
alh = blh + clh;
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	00101bbb001001000 BBBCCCCCAAAAAA
VADD4B	a,b,u6	00101bbb011001000 BBBuuuuuuAAAAAA
VADD4B	b,b,s12	00101bbb101001000 BBBssssssSSSSSS
VADD4B<.cc>	b,b,c	00101bbb111001000 BBBCCCCC0QQQQQ
VADD4B<.cc>	b,b,u6	00101bbb111001000 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb000101001BBBCCCCC <del>AAAAAA</del>
VADDS2H	a,b,u6	00101bbb010101001BBBuuuuuu <del>AAAAAA</del>
VADDS2H	b,b,s12	00101bbb100101001BBBsssss <del>SSSSSS</del>
VADDS2H<.cc>	b,b,c	00101bbb110101001BBBCCCCC <del>0QQQQQ</del>
VADDS2H<.cc>	b,b,u6	00101bbb110101001BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#).

## VADDSSUB2H

### 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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VADDSSUB2H a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

bl = b & 0x0000_ffff;                                /* VADDSSUB2H */
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
al = bl + cl;
ah = bh - ch;
a = (al & 0x0000_ffff) | (ah << 16);

```

## Assembly Code Example

```
VADDSSUB2H r1,r2,r3      ; dual SIMD 16-bit addition and subtraction
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page 80. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page 82 and “[Expansion of Literals](#)” on page 82.

### Instruction Code

VADDSSUB2H	a,b,c	00101bbb000101100 BBBCCCCC AAAAAAA
VADDSSUB2H	a,b,u6	00101bbb010101100 BBBuuuuuu AAAAAAA
VADDSSUB2H	b,b,s12	00101bbb100101100 BBBssssss SSSSSS
VADDSSUB2H<.cc>	b,b,c	00101bbb110101100 BBBCCCCC 0QQQQQ
VADDSSUB2H<.cc>	b,b,u6	00101bbb110101100 BBBuuuuuu 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:

```
ADD.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](#).

## 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:

```
ADD .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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VADDSUBS2H a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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 are 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

```

bl = b & 0x0000_ffff;                                /* VADDSSUBS2H */
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 |= s

```

## Assembly Code Example

```

VADDSSUBS2H 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](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

### Instruction Code

VADDSSUBS2H	a, b, c	00101bbb000101101BBBCCCCC <del>AAAAAA</del>
VADDSSUBS2H	a, b, u6	00101bbb010101101BBBuuuuuu <del>AAAAAA</del>
VADDSSUBS2H	b, b, s12	00101bbb100101101BBBsssss <del>SSSSSS</del>
VADDSSUBS2H<.cc>	b, b, c	00101bbb110101101BBBCCCCC <del>0QQQQQ</del>
VADDSSUBS2H<.cc>	b, b, u6	00101bbb110101101BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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.h0 = b.h0                                /* VALGN2H */
a.h1 = c.h1;
```

## 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	00101bbb000011010BBBCCCCC <del>AAAAAA</del>
VALGN2H	a,b,u6	00101bbb010011010BBBuuuuuu <del>AAAAAA</del>
VALGN2H	b,b,s12	00101bbb100011010BBBssssss <del>SSSSSS</del>
VALGN2H<.cc>	b,b,c	00101bbb110011010BBBCCCCC <del>0QQQQQ</del>
VALGN2H<.cc>	b,b,u6	00101bbb110011010BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb001000010 BBBCCCCCAAAAAA
VASL2H	a,b,u6	00101bbb011000010 BBBuuuuuuAAAAAA
VASL2H	b,b,s12	00101bbb101000010 BBBssssssSSSSSS
VASL2H<.cc>	b,b,c	00101bbb111000010 BBBCCCCC0QQQQQ
VASL2H<.cc>	b,b,u6	00101bbb111000010 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

bl = b & 0x0000_ffff;
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

```

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	00101bbb001000011BBBCCCCC <del>AAAAAA</del>
VASLS2H	a,b,u6	00101bbb011000011BBBuuuuuu <del>AAAAAA</del>
VASLS2H	b,b,s12	00101bbb101000011BBBsssss <del>SSSSSS</del>
VASLS2H<.cc>	b,b,c	00101bbb111000011BBBCCCCC <del>0QQQQQ</del>
VASLS2H<.cc>	b,b,u6	00101bbb111000011BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb001000100 BBBCCCCCAAAAAA
VASR2H	a,b,u6	00101bbb011000100 BBBuuuuuuAAAAAA
VASR2H	b,b,s12	00101bbb101000100 BBBssssssSSSSSS
VASR2H<.cc>	b,b,c	00101bbb111000100 BBBCCCCC0QQQQQ
VASR2H<.cc>	b,b,u6	00101bbb111000100 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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;
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	00101bbb101000101BBBsssssSSSSSS
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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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;
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	00101bbb101000111BBBsssssSSSSSS
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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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	00101bbb001011110BBBCCCCC100100
VEXT2BHL	b,limm	00101bbb001011110BBB111110100100
VEXT2BHL	0,c	00101110001011110111CCCCCC100100
VEXT2BHL	0,limm	00101110001011110111111110100100
VEXT2BHL	b,u6	00101bbb011011110BBBuuuuuu100100
VEXT2BHL	0,u6	00101110011011110111uuuuuu100100

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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 <b>bbb</b> 001011110 <b>BBB</b> CCCCCC100000
VEXT2BHLF	b,u6	00101 <b>bbb</b> 011011110 <b>BBB</b> uuuuuuu100000

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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	00101bbb001011110BBBCCCCC100101
VEXT2BHM	b,limm	00101bbb001011110BBB111110100101
VEXT2BHM	0,c	00101110001011110111CCCCCC100101
VEXT2BHM	0,limm	00101110001011110111111110100101
VEXT2BHM	b,u6	00101bbb011011110BBBuuuuuu100101
VEXT2BHM	0,u6	00101110011011110111uuuuuu100101

## 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:

```
ADD.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

VEXT2BHM**F** b,c

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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)); /* VEXT2BHM*/
```

## Assembly Code Example

```
VEXT2BHM r1,r2 ; Expand and pack 2 lower bytes from r1  
; into 2 16b words in r0
```

## Syntax and Encoding

### Instruction Code

VEXT2BHM	b,c	00101 <b>bbb</b> 001011110 <b>BBB</b> CCCCCC100001
VEXT2BHM	b,u6	00101 <b>bbb</b> 011011110 <b>BBB</b> uuuuuu100001

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb001000110 BBBCCCCCAAAAAA
VLSR2H	a,b,u6	00101bbb011000110 BBBuuuuuuAAAAAA
VLSR2H	b,b,s12	00101bbb101000110 BBBssssssSSSSSS
VLSR2H<.cc>	b,b,c	00101bbb111000110 BBBCCCCC0QQQQQ
VLSR2H<.cc>	b,b,u6	00101bbb111000110 BBBuuuuuu1QQQQQ

## 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:

```
ADD.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

DSP Vector 16x16 MAC

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 writeback.

## 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	00101bbb000111100BBBCCCCC <del>AAAAAA</del>
VMAC2H	a,b,u6	00101bbb010111100BBBuuuuuu <del>AAAAAA</del>
VMAC2H	b,b,s12	00101bbb100111100BBBssssss <del>SSSSSS</del>
VMAC2H<.cc>	b,b,c	00101bbb110111100BBBCCCCC <del>0QQQQQ</del>
VMAC2H<.cc>	b,b,u6	00101bbb110111100BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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;
    sl      = 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;
    sl      = 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 |= sl | 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb000100010BBBCCCCC <del>AAAAAA</del>
VMAC2HNFR	a,b,u6	00110bbb010100010BBBuuuuuu <del>AAAAAA</del>

VMAC2HNFR	b, b, s12	00110 <b>bbb</b> 100100010 <b>BBB</b> ssssssSSSSSS
VMAC2HNFR<.cc>	b, b, c	00110 <b>bbb</b> 110100010 <b>BBB</b> CCCCCC0QQQQQ
VMAC2HNFR<.cc>	b, b, u6	00110 <b>bbb</b> 110100010 <b>BBB</b> uuuuuu1QQQQQ

## 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:

```
ADD.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

DSP Vector 16x16 MAC

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 writeback.

## 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	00101bbb000111110BBBCCCCC <del>AAAAAA</del>
VMAC2HU	a,b,u6	00101bbb010111110BBBuuuuuu <del>AAAAAA</del>
VMAC2HU	b,b,s12	00101bbb100111110BBBssssss <del>SSSSSS</del>
VMAC2HU<.cc>	b,b,c	00101bbb110111110BBBCCCCC <del>0QQQQQ</del>
VMAC2HU<.cc>	b,b,u6	00101bbb110111110BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb001001001BBBCCCCC <del>AAAAAA</del>
VMAX2H	a,b,u6	00101bbb011001001BBBuuuuuu <del>AAAAAA</del>
VMAX2H	b,b,s12	00101bbb101001001BBBssssss <del>SSSSSS</del>
VMAX2H<.cc>	b,b,c	00101bbb111001001BBBCCCCC <del>0QQQQQ</del>
VMAX2H<.cc>	b,b,u6	00101bbb111001001BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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	00101bbb001001011BBBCCCCC <del>AAAAAA</del>
VMIN2H	a,b,u6	00101bbb011001011BBBuuuuuu <del>AAAAAA</del>
VMIN2H	b,b,s12	00101bbb101001011BBBssssss <del>SSSSSS</del>
VMIN2H<.cc>	b,b,c	00101bbb111001011BBBCCCCC <del>0QQQQQ</del>
VMIN2H<.cc>	b,b,u6	00101bbb111001011BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

DSP Vector 16x16 MAC

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 writeback.

## 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](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

Instruction Code		
VMPY2H	a, b, c	00101bbb000111000BBBCCCCC <del>AAAAAA</del>
VMPY2H	a, b, u6	00101bbb010111000BBBuuuuuu <del>AAAAAA</del>
VMPY2H	b, b, s12	00101bbb100111000BBBssssss <del>SSSSSS</del>
VMPY2H<.cc>	b, b, c	00101bbb110111000BBBCCCCC <del>0QQQQQ</del>
VMPY2H<.cc>	b, b, u6	00101bbb110111000BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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_ff;
        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_ff;
        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_ff;
        sl   = sat(acclo, 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_ff;
        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;
    sl      = 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;
    sl      = 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 |= sl | 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&lt;.cc&gt;

b, b, u6

00101bbb110111001BBBuuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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;
    sl      = 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;
    sl      = 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 |= sl | 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	00101bbb100111011BBBsssssSSSSSS
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:

```
ADD.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

DSP Vector 16x16 MAC

### 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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 writeback.

## 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](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

### Instruction Code

VMPY2HU	a,b,c	00101bbb000111010BBBCCCCC <del>AAAAAA</del>
VMPY2HU	a,b,u6	00101bbb010111010BBB <u>uuuuuuAAAAAA</u>
VMPY2HU	b,b,s12	00101bbb100111010BBB <del>s</del> ssssssSSSSSS
VMPY2HU<.cc>	b,b,c	00101bbb110111010BBBCCCCC <del>0</del> QQQQQ
VMPY2HU<.cc>	b,b,u6	00101bbb110111010BBB <u>uuuuuu1QQQQQ</u>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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 writeback.

## 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;
    sl      = 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;
    sl      = 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 |= sl | 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	00101bbb011000000BBBuuuuuuuAAAAAA
VMPY2HWF	b,b,s12	00101bbb101000000BBBssssssSSSSSS
VMPY2HWF<.cc>	b,b,c	00101bbb111000000BBBCCCCC0QQQQQ
VMPY2HWF<.cc>	b,b,u6	00101bbb111000000BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#).

## 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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VMSUB2HF a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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	00110bbb00001000BBBCCCCC <del>AAAAAA</del>
VMSUB2HF	a,b,u6	00110bbb010001000BBBuuuuuu <del>AAAAAA</del>
VMSUB2HF	b,b,s12	00110bbb100001000BBBsssssSSSSSS
VMSUB2HF<.cc>	b,b,c	00110bbb110001000BBBCCCCC0QQQQQ
VMSUB2HF<.cc>	b,b,u6	00110bbb110001000BBBuuuuuu1QQQQQ

## 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:

```
ADD.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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VMSUB2HFR a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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;
    sl   = 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;
    sl   = 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 |= sl | 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	00110bbb00000110 BBBCCCCC <del>AAAAAA</del>
VMSUB2HFR	a,b,u6	00110bbb010000110 BBB <u>uuuuuuAAAAAA</u>
VMSUB2HFR	b,b,s12	00110bbb100000110 BBB <u>ssssssSSSSSS</u>
VMSUB2HFR<.cc>	b,b,c	00110bbb110000110 BBBCCCCC <del>0QQQQQ</del>
VMSUB2HFR<.cc>	b,b,u6	00110bbb110000110 BBB <u>uuuuuu1QQQQQ</u>

## 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:

```
ADD.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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VMSUB2HNFR a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for implicit accumulator update; three cycle latency for explicit writeback

### 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;                                                       VMSUB2HNFR*/
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	00110bbb000100011BBBCCCCC <del>AAAAAA</del>
VMSUB2HNFR	a,b,u6	00110bbb010100011BBBuuuuuu <del>AAAAAA</del>

VMSUB2HNFR	b, b, s12	00110 <b>bbb</b> 100100011 <b>BBB</b> ssssssSSSSSS
VMSUB2HNFR<.cc>	b, b, c	00110 <b>bbb</b> 110100011 <b>BBB</b> CCCCCC0QQQQQ
VMSUB2HNFR<.cc>	b, b, u6	00110 <b>bbb</b> 110100011 <b>BBB</b> uuuuuu1QQQQQ

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
bl = 0 - cl;
bh = 0 - ch;
b = (bl & 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	00101bbb001011110BBBCCCCC101010
VNEG2H	b,limm	00101bbb001011110BBB111110101010
VNEG2H	0,c	00101110001011110111CCCCCC101010
VNEG2H	0,limm	00101110001011110111111110101010
VNEG2H	b,u6	00101bbb011011110BBBuuuuuu101010
VNEG2H	0,u6	00101110011011110111uuuuuu101010

## 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:

ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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;
ch = (c >> 16) & 0x0000_ffff;
sl = sat((sint128)0 - c1, 15, &bl);
sh = sat((sint128)0 - ch, 15, &bh);
DSP_CTRL.SAT |= sl | sh;
b = (bl & 0x0000_ffff) | (bh << 16);
/* VNEGS2H */

```

## Assembly Code Example

```
VNEGS2H r1,r2 ; Negate vector r2 and return saturated result in r1
```

## Syntax and Encoding

Instruction Code		
VNEGS2H	b,c	00101bbb00101110BBBCCCCC101011
VNEGS2H	b,limm	00101bbb00101110BBB111110101011
VNEGS2H	0,c	0010111000101110111CCCCC101011
VNEGS2H	0,limm	0010111000101110111111110101011
VNEGS2H	b,u6	00101bbb01101110BBBuuuuuu101011
VNEGS2H	0,u6	0010111001101110111uuuuuu101011

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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;
}
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
bl = normh(cl);
bh = normh(ch);
b = (bl & 0x0000_ffff) | (bh << 16);

```

## 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	00101bbb001011100BBBCCCCC101100
VNORM2H	b,limm	00101bbb001011110BBB111110101100
VNORM2H	0,c	00101110001011110111CCCCCC101100
VNORM2H	0,limm	00101110001011110111111110101100
VNORM2H	b,u6	00101bbb01101110BBBuuuuuuu101100
VNORM2H	0,u6	00101110011011110111uuuuuu101100

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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	00101bbb001010010BBBCCCCC <del>AAAAAA</del>
VPACK2HL	a, b, u6	00101bbb011010010BBBuuuuuu <del>AAAAAA</del>
VPACK2HL	b, b, s12	00101bbb101010010BBBssssss <del>SSSSSS</del>
VPACK2HL<.cc>	b, b, c	00101bbb111010010BBBCCCCC0 <del>QQQQQ</del>
VPACK2HL<.cc>	b, b, u6	00101bbb111010010BBBuuuuuu1 <del>QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

Compose the destination operand from the higher16-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	00101bbb001010011BBBCCCCC <del>AAAAAA</del>
VPACK2HM	a, b, u6	00101bbb011010011BBBuuuuuu <del>AAAAAA</del>
VPACK2HM	b, b, s12	00101bbb101010011BBBssssss <del>SSSSSS</del>
VPACK2HM<.cc>	b, b, c	00101bbb111010011BBBCCCCC0 <del>QQQQQ</del>
VPACK2HM<.cc>	b, b, u6	00101bbb111010011BBBuuuuuu1 <del>QQQQQ</del>

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

### Description

Extract the two most significant bytes from 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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	00101bbb001011110BBBCCCCC011111
VPACK2HBMF	b,u6	00101bbb011011110BBBuuuuuu011111

## 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:

```
ADD.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 =vperm(b, c)`

### Instruction Format

`op a,b,c`

### Syntax Example

`VPERM a,b,c`

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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 writeback the result in r0

```

## Syntax and Encoding

### Instruction Code

VPERM	a,b,c	00101bbb001011100BBBCCCCC <del>AAAAAA</del>
VPERM	a,b,u6	00101bbb011011100BBBuuuuuu <del>AAAAAA</del>

VPERM	b, b, s12	00101bbb101011100BBBssssssSSSSSS
VPERM<.cc>	b, b, c	00101bbb111011100BBBCCCCC0QQQQQ
VPERM<.cc>	b, b, u6	00101bbb111011100BBBuuuuuu1QQQQQ

## 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:

```
ADD.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-significant16 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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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	00101110001011110111CCCCCC100010
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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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	00101110001011110111CCCCCC100011
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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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	00101110001011110111CCCCCC100110
VSEXT2BHL	0,limm	00101110001011110111111110100110
VSEXT2BHL	b,u6	00101bbb011011110BBBuuuuuuu100110
VSEXT2BHL	0,u6	00101110011011110111uuuuuuu100110

## 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:

```
ADD.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

### Timing Characteristics

Issue one instruction per cycle; one cycle latency for result writeback

### 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	00101bbb001011110BBBCCCCC100111
VSEXT2BHM	b,limm	00101bbb001011110BBB111110100111
VSEXT2BHM	0,c	00101110001011110111CCCCCC100111
VSEXT2BHM	0,limm	00101110001011110111111110100111
VSEXT2BHM	b,u6	00101bbb011011110BBBuuuuuuu100111
VSEXT2BHM	0,u6	00101110011011110111uuuuuuu100111

## 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:

```
ADD.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

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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VSUB2H a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

bl = b & 0x0000_ffff;                                /* VSUB2H */
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
al = bl - cl;
ah = bh - ch;
a = (al & 0x0000_ffff) | (ah << 16);

```

## Assembly Code Example

```
VSUB2H r1,r2,r3 ; dual SIMD 16-bit subtraction
```

## Syntax and Encoding



For detailed information about vector operands, see “[Vector Operands](#)” on page 80. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page 82 and “[Expansion of Literals](#)” on page 82.

### Instruction Code

VSUB2H	a,b,c	00101bbb000101010BBBCCCCC <del>AAAAAA</del>
VSUB2H	a,b,u6	00101bbb010101010BBBuuuuuu <del>AAAAAA</del>
VSUB2H	b,b,s12	00101bbb100101010BBBsssss <del>SSSSSS</del>
VSUB2H<.cc>	b,b,c	00101bbb110101010BBBCCCCC <del>0QQQQQ</del>
VSUB2H<.cc>	b,b,u6	00101bbb110101010BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VSUBS2H a,b,c

### Timing Characteristic

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

bl = b & 0x0000_ffff;                                /* VSUBS2H */
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 |= 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	00101bbb000101011BBBCCCCC <del>AAAAAA</del>
VSUBS2H	a,b,u6	00101bbb010101011BBBuuuuuu <del>AAAAAA</del>
VSUBS2H	b,b,s12	00101bbb100101011BBBsssss <del>SSSSSS</del>
VSUBS2H<.cc>	b,b,c	00101bbb110101011BBBCCCCC <del>0QQQQQ</del>
VSUBS2H<.cc>	b,b,u6	00101bbb110101011BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## VSUBS4H

### Function

Four-way 16-bit vector subtraction. The result is saturated.

### Extension Group

HAS\_DSP == 1

### Operation

```
if (cc) {
    A.h0 = SAT16(B.h0 - C.h0);
    A.h1 = SAT16(B.h1 - C.h1);
    A.h2 = SAT16(B.h2 - C.h2);
    A.h3 = SAT16(B.h3 - C.h3);
}
```



**Note** h0, h1, h2, and h3 represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see “Data Formats” on page 80.

### Instruction Format

op a, b, c

### Syntax Example

VSUBS4H 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 four 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

```
s = sat((sint128)b.h0 - c.h0, 15, &a.h0) | /* VSUBS4H*/
    sat((sint128)b.h1 - c.h1, 15, &a.h1) |
    sat((sint128)b.h2 - c.h2, 15, &a.h2) |
    sat((sint128)b.h3 - c.h3, 15, &a.h3);
DSP_CTRL.SAT |= s;
```

## Assembly Code Example

```
VSUBS4H r0,r2,r4      ; Subtract four vectors {r2,r3} and {r4,r5} and
                        ; return saturated result in {r0,r1}
```

## Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” on page [258](#) and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” on page [259](#).

Instruction Code		
VSUBS4H	a,b,c	00101bbb001110011 BBBCCCCC <del>AAAAAA</del>
VSUBS4H	a,b,u6	00101bbb011110011 BBBuuuuuu <del>AAAAAA</del>
VSUBS4H	b,b,s12	00101bbb101110011 BBBssssss <del>SSSSSS</del>
VSUBS4H<.cc>	b,b,c	00101bbb111110011 BBBCCCCC <del>0QQQQQ</del>
VSUBS4H<.cc>	b,b,u6	00101bbb111110011 BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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

```
a.b3 = b.b3-c.b3;
a.b2 = b.b2-c.b2;
a.b1 = b.b1-c.b1;
a.b0 = b.b0-c.b0;
```



**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

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

bll = b & 0x0000_00ff;                                /* VSUB4B */
blh = b & 0x0000_ff00;
bhl = b & 0x00ff_0000;
bhh = b & 0xff00_0000;
c1l = c & 0x0000_00ff;
clh = c & 0x0000_ff00;
chl = c & 0x00ff_0000;
chh = c & 0xff00_0000;
all = bll - c1l;
alh = blh - clh;
ahl = bhl - chl;
ahh = bhh - chh;
a = (all & 0x0000_00ff) | (alh & 0x0000_ff00) | (ahl & 0x00ff_0000) |
     ahh;

```

## 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	00101bbb001001010BBBCCCCCAAAAAA
VSUB4B	a,b,u6	00101bbb011001010BBBuuuuuuAAAAAA
VSUB4B	b,b,s12	00101bbb101001010BBBssssssSSSSSS
VSUB4B<.cc>	b,b,c	00101bbb111001010BBBCCCCC0QQQQQ
VSUB4B<.cc>	b,b,u6	00101bbb111001010BBBuuuuuu1QQQQQ

## 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:

```
ADD.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

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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VSUBADD2H a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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 you use the loop-counter (LP\_COUNT/r60) as a destination operand register.

## Pseudo Code

```

bl = b & 0x0000_ffff;                                /* VSUBADD2H */
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
al = bl - cl;
ah = bh + ch;
a = (al & 0x0000_ffff) | (ah << 16);

```

## 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](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

### Instruction Code

VSUBADD2H	a,b,c	00101bbb000101110BBBCCCCC <del>AAAAAA</del>
VSUBADD2H	a,b,u6	00101bbb010101110BBBuuuuuu <del>AAAAAA</del>
VSUBADD2H	b,b,s12	00101bbb100101110BBBsssss <del>SSSSSS</del>
VSUBADD2H<.cc>	b,b,c	00101bbb110101110BBBCCCCC <del>0QQQQQ</del>
VSUBADD2H<.cc>	b,b,u6	00101bbb110101110BBBuuuuuu <del>1QQQQQ</del>

## 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:

```
ADD.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](#).

## 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:

```
ADD.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](#)” on page [80](#).

### Instruction Format

op a, b, c

### Syntax Example

VSUBADDS2H a,b,c

### Timing Characteristics

Issue one instruction per cycle; two cycle latency for result writeback

### 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

```

bl = b & 0x0000_ffff;                                /* VSUBADDS2H */
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 |= 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](#)” on page [80](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” on page [82](#) and “[Expansion of Literals](#)” on page [82](#).

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:

```
ADD.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](#).

## 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:

```
ADD.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

# Appendices

## In this part:

- Build and Auxiliary Register List
- ISA Version Differences
- Implementation-dependent Behavior
- Auxiliary Functions for DSP and XY Extensions
- Processor Component Options



# A

## Build and Auxiliary Register List

This chapter lists all the auxiliary registers and build configuration registers available in the ARC EM family of processors. A discussion about the baseline auxiliary and build configuration registers are discussed in the [Core Architectural State Registers](#). The auxiliary and build configuration registers for each system or memory component are discussed in that respective component chapter.

**Table A-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
0x18	DCCM Base Address, AUX_DCCM
0x19	Invalidate Instruction-Cache Line, IC_IVIL
0x1A	Instruction Cache External-Access Address, IC_RAM_ADDR
0x1B	Instruction-Cache Tag Access, IC_TAG
0x1C	Instruction Cache Secure Bit Tag Register, IC_XTAG

**Table A-1 Auxiliary Register List (Continued)**

Register Address	Register Name
0x1D	Instruction Cache Data Access, IC_DATA
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
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
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

**Table A-1 Auxiliary Register List (Continued)**

Register Address	Register Name
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
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
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

**Table A-1 Auxiliary Register List (Continued)**

Register Address	Register Name
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
0x238	Watchdog Password Register, WDT_PASSWD
0x239	Watchdog Timer Register, WDT_CTRL
0x23A	Watchdog Timer Period Register, WDT_PERIOD
0x23B	Watchdog Timer Count Register, WDT_COUNT
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, STACK_TOP
0x261	User Stack Region Base Address, STACK_BASE
0x262	Secure User Mode Secure Stack Region Top Address, S_STACK_TOP
0x263	Secure User Mode Secure Stack Region Base Address, S_STACK_BASE
0x264	Kernel Stack Region Top Address, STACK_TOP
0x265	Kernel Stack Region Base Address, STACK_BASE
0x266	Secure Kernel Mode Secure Stack Region Top Address, S_STACK_TOP

**Table A-1 Auxiliary Register List (Continued)**

Register Address	Register Name
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	NVM ICCM Control Register, NV_ICCM_CTRL
0x355	NVM ICCM Erase Register, NV_ICCM_ERASE
0x356	NVM ICCM Programming Register, NV_ICCM_PROG
0x357	NVM 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 RTT Address Register, RTT_ADDRESS
0x381	ARC RTT DATA Register, RTT_DATA
0x382	ARC RTT CMD Register, RTT_COMMAND
0x400	Exception Return Address, ERET
0x401	Exception Return Branch Target Address, ERBTA
0x402	Exception Return Status, ERSTATUS
0x403	Exception Cause Register, ECR
0x404	Exception Fault Address, EFA

**Table A-1 Auxiliary Register List (Continued)**

Register Address	Register Name
0x405	ECC Syndrome Register, EYSN
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_RENDER
0x44B	MPU Region Permission Register, MPU_RPER
0x44C	MPU Probe Register, MPU_PROBE
0x44F	User Extension Flags Register, XFLAGS
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

**Table A-1 Auxiliary Register List (Continued)**

Register Address	Register Name
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, MCIP_CMD
0x601	ARConnect Write Data Register, MCIP_WDATA
0x602	ARConnect Read Data Register, MCIP_READBACK
0x610	Core Power Status Register, PDM_PSTAT
0x611	ARC RTT Power Status Register, RTT_PDM_PSTAT
0x613	Power Down Register, PDM_PMODE
0x680	DMA Controller Configuration Register, DMACTRL
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
0x690	DMA Channel Control Register, DMACTRLx
0x691	DMA Channel Source Address Register, DMASARx
0x692	DMA Channel Destination Address Register, DMADARx

**Table A-1 Auxiliary Register List (Continued)**

Register Address	Register Name
Memory-mapped	DMA Channel Linked-List Pointer Register, DMALLPX
0x700	SmaRT Control Register, SMART_CONTROL
0x701	SmaRT Data Register, SMART_DATA
0xA80	Lockstep Control Register, LSC_CTRL
0xA81	Lockstep Debug Control Register, LSC_MCD
0xA82	Lockstep Status Register, LSC_STATUS
0xA83	Lockstep Error Register, LSC_ERROR_REG

## A.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 Only Access](#)).

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

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

**Table A-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

**Table A-2 Build Configuration Registers (Continued)**

Number	Name
0x6D	Memory Protection Build Configuration Register, MPU_BUILD
0x6E	Core Register File Configuration Register, RF_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
0x79	XY Build Configuration Register, XY_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
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, MCIP_SYSTEM_BUILD
0xD1	Inter-core Semaphore Unit Build Configuration Register, MCIP_SEMA_BUILD
0xD2	Inter-core Message Unit Build Configuration Register, MCIP_MESSAGE_BUILD
0xD3	Power Management Unit Build Configuration Register, MCIP_PMU_BUILD
0xD6	ARConnect Global Free Running Counter Build Configuration Register, MCIP_GFRC_BUILD

**Table A-2 Build Configuration Registers (Continued)**

Number	Name
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 Build Configuration Register, MCIP_ICI_BUILD
0xE1	Inter-Core Debug Unit Build Configuration Register, MCIP_ICD_BUILD
0xE3	ARConnect Power Domain Management Build Configuration Register, MCIP_PDM_BUILD
0xEF	Lockstep Configuration Register, LSC_BUILD_AUX
0xF1	Core Architecture Build Configuration Register, ARCH_CONFIG
0xF2	ARC RTT 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

**B**

# 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 family of processors include the EFA register only if required by the inclusion of MMU or MPU options. The EFA register is always included in the 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 B-1](#) and [Table B-2](#) describe the known differences in the baseline and optional architecture features between ARCv2, ARCompact for ARC 600, and ARCompact for ARC 700.

**Table B-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	ARC EM family: Optional ARC HS family: Baseline

**Table B-1 Baseline Differences: ARC 600, ARC 700, and ARCv2 (Continued)**

Baseline ISA Features	ARC 600	ARC 700	ARCv2
STATUS register (obsolete)	Yes Not in 601	Yes	No
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 LIMM 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

**Table B-1 Baseline Differences: ARC 600, ARC 700, and ARCv2 (Continued)**

<b>Baseline ISA Features</b>	<b>ARC 600</b>	<b>ARC 700</b>	<b>ARCv2</b>
Use of undefined condition codes	Always returns FALSE	Instruction Error exception	Instruction Error exception
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 B-2 ISA Option Differences: ARC 600, ARC 700, and ARCv2**

<b>Optional ISA Features</b>	<b>ARC 600</b>	<b>ARC 700</b>	<b>ARCv2</b>
EX	No	Baseline	Baseline
EFA auxiliary register	No	Baseline	Optional
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

**Table B-2 ISA Option Differences: ARC 600, ARC 700, and ARCv2 (Continued)**

Optional ISA Features	ARC 600	ARC 700	ARCv2
MPYW MPYUW	Optional	No	Optional
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 B-3 ISA Features: ARC EM and ARC HS**

<b>ISA Feature</b>	<b>ARC EM</b>	<b>ARC HS</b>
Multiply and accumulate extension instructions: QMPYH, QMPYHU, DMPYWH, DMPYWHU, QMACH, QMACHU, DMACWH, DMACWHU, VADD4H, VSUB4H, VADDSUB4H, VSUBADD4H, VADD2, VSUB2, VADDSUB, VSUBADD	No	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
DSP Extension Instructions for Complex Numbers	Optional Select these instructions using the DSP_COMPLEX==1	No
LPcc: branch or jump in last position within loop body	Affects loop count	Disallowed: triggers illegal instruction sequence exception (see LPcc instruction)
Double-word 64-bit operations	No	Optional, Enable/disable 64-bit operations using LL64_OPTION = {0,1}. LDD and STD are load and store operations for 64-bit data. When atomic operation are also enabled, LLOCKD and SCOND instructions are enabled.
r58 and r59	Extension core registers When HAS_DSP==1, 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 FSMAADD and FSMSUB instructions are enabled, the two extension core registers (r58 and r59) are included and comprise the third operand.	When MPY_OPTION > 6, 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 MPY_OPTION > 6 or FPU_FMA_OPTION is defined, two extension core registers (r58 and r59) are included and comprise the 64-bit Accumulator (ACCH, ACCL).

**Table B-3 ISA Features: ARC EM and ARC HS**

<b>ISA Feature</b>	<b>ARC EM</b>	<b>ARC HS</b>
IFQUEUE_BUILD auxiliary register	Included in this ISA and configurable.	Not included in this ISA.
CODE_DENSITY_OPTION	Optional	Yes
BIT_SCAN_OPTION	Optional	Yes
SWAP_OPTION	Optional	Yes
ATOMIC_OPTION	No	Optional; When ATOMIC_OPTION==1, the following instructions are supported: LLOCK, SCOND, WLFC Further when LL64_OPTION==1, the LLOCKD and SCOND instructions are supported.
Floating-point instructions	Yes	Yes
IC FEATURE LEVEL	Optional; allowed values are 0, 1, 2	Fixed at 2.
DC FEATURE LEVEL	Optional; allowed values are 0, 1, 2	Fixed at 2.
Degree of associativity for data caches (DC_WAYS)	Optional; allowed values are 0, 1, 2	Fixed at 2.
Support for Memory management Unit	No	Yes
Support for L2 cache	No	Yes
Baseline instructions: PREFETCHW, DSYNC, DMB	No	Yes
Support for Shared Coherency Unit	No	Yes
Support for the Vector Unit	No	Yes
Multi-core cluster component	No	Yes
Bitstream instructions	Yes	No
XY instructions: MODIF, MODAPP	Yes	No



# C

## Implementation-dependent Behavior

### C.1 EM Exception Handling

#### C.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 prologue or epilogue pushes or pops registers to or from the kernel stack when a user-mode process is interrupted. Kernel privileges are applied to all interrupt prologue memory operations that access the kernel stack. Upon completion of register push operations during an interrupt prologue, the processor transitions into kernel mode, but the processor technically remains in its pre-interrupt Kernel/User mode until the completion of the prologue.

### C.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 fills 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

## C.3 Exception Handling

### C.3.1 ECR User-mode Setting for the ARC EM 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 prologue or epilogue 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 will still reflect the committed value of STATUS32.U, and will not be 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.

## C.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. Hence, the result of such a division operator should not be relied upon by software.

**Table C-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		Write 0 to the destination register

# D

## Processor Component Options

### D.1 COMPONENT: com.arc.hardware.Actionpoints.1\_0

```

Option:          -num_actionpoints
Description:    This is the number of trigger events available.
Units:          bytes
Default value:  2
Possible values: 2, 4, 8

Option:          -aps_feature
Description:    Selects Actionpoint feature set
Units:
Default value:  min
Possible values: min, full

```

### D.2 COMPONENT: com.arc.hardware.AGU.1\_0

```

Option:          -agu_size
Description:    Predefined configurations of modifiers, address
pointers and offset registers
<pre>

      address      address
      pointers     offset regs   modifiers
      -----      -----
small:       4           2           4
medium:     8           4          12
large:      12          8          24
</pre>

Units:
Default value: small

```

Possible values: small, medium, large

Option: -agu\_accord  
Description: Enable the accordion stage if operating frequency is critical  
Units:  
Default value: true  
Possible values: true, false

Option: -agu\_wb\_depth  
Description: Write buffer depth  
Units: bytes  
Default value: 2  
Possible values: 2, 4

=====

### D.3 COMPONENT: com.arc.hardware.AR Cv2EM.1\_0

Option: -arcv2em  
Description: Description to follow  
Units:  
Default value: true  
Possible values: true, false

Option: -def\_div2ref  
Description: This specifies the clock division factor at reset. It is used for mss clock controller to generate core clock, and the value N means core is running at (1/N) x ref\_clk.  
Units: bytes  
Default value: 1  
Minimum value: 1  
Maximum value: 255

Option: -addr\_size  
Description: This defines the address bus width (in bits).  
Units: bytes  
Default value: 16  
Possible values: 16, 20, 24, 32

Option: -pc\_size  
Description: This defines the program counter (in bits).  
Units: bytes  
Default value: 16  
Possible values: 16, 24, 32

Option: -lpc\_size  
Description: This defines the size of the loop counter (in bits).  
Units: bytes  
Default value: 8  
Possible values: 8, 16, 32

Option: -halt\_on\_reset

Description: This defines whether the core is halted initially on reset.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -byte\_order  
 Description: This defines the endianness of the core.  
 Units:  
 Default value: little  
 Possible values: little, big

Option: -code\_density\_option  
 Description: This reduces the size of program memory by adding instructions that condense commonly used instruction patterns with some marginal increase in processor gate count. The added instructions are ENTER\_S, LEAVE\_S, JLI\_S, BI, BIH.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -bitscan\_option  
 Description: This adds instructions for efficient search of bits within a 32 bit word, including normalize (NORM, NORMH, NORMW) and find first or last set bit (FFS, FLS) instructions.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -shift\_option  
 Description: The Shift ISA option adds variable and multi-length shift rotation instructions: (0) No shift/rotation instructions (1) ASR16, ASR8, LSR8, LSL8, ROL8, ROR8 (2) ASRM, ASLM, LSRM, RORM (3) ASR16, ASR8, LSR8, LSL8, ROL8, ROR8, ASRM, ASLM, LSRM, RORM  
 Units: bytes  
 Default value: 0  
 Possible values: 0, 1, 2, 3

Option: -swap\_option  
 Description: This adds two instructions used to swap half-words or bytes in a 32b word. Useful for converting between little to big endianess and vice-versa.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -div\_rem\_option  
 Description: The DIV/REM option adds non-blocking multi-cycle implementation of integer divide/remainder functions. Added instructions are DIV, DIVU (integer divide), REM and REMU (integer divide remainder). radix2 takes 33 cycles. radix4\_enhanced takes 3 to 19 cycles per operation.  
 Units:  
 Default value: none  
 Possible values: none, radix2, radix4\_enhanced

Option: -mpy\_option

**Description:** The Multiplier ISA option allows selection between several multiplier configurations to tradeoff performance with silicon area. For select multiply options, when the DIV/REM option is also selected, some datapath resources will be shared between the multiply and divide pipeline to minimize total area.

Cycle count (16-bit, lower 32-bit or upper 32-bit) for the different configurations is as follows:

<pre>

| option | 16/L32/U32 | Instructions |
|--------|------------|--------------|
|--------|------------|--------------|

|      |       |                       |
|------|-------|-----------------------|
| none | -/-/- | None                  |
| wlh1 | 1/1/1 | MPYW/U, MPY/U, MPYH/U |
| wlh2 | 2/2/2 | MPYW/U, MPY/U, MPYH/U |
| wlh3 | 2/3/3 | MPYW/U, MPY/U, MPYH/U |
| wlh4 | 2/4/5 | MPYW/U, MPY/U, MPYH/U |
| wlh5 | 5/9/9 | MPYW/U, MPY/U, MPYH/U |

</pre>

Units:

Default value: wlh5

Possible values: none, wlh1, wlh2, wlh3, wlh4, wlh5

Option: -code\_protection

**Description:** The ARC EM 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.

Units:

Default value: false

Possible values: true, false

Option: -stack\_checking

**Description:** Stack checking is a mechanism for checking stack accesses and raising an exception when a stack overflow or underflow is detected.

Units:

Default value: false

Possible values: true, false

Option: -unaligned\_option

**Description:** This enables unaligned loads and stores.

Units:

Default value: false

Possible values: true, false

Option: -intvbase\_preset

**Description:** This sets the interrupt vector base configuration register, VECBASE\_AC\_BUILD. The vector base address is aligned to a 1KB boundary, so the required address value should be divided by 1K (i.e. do not include the lower 10 bits). On reset,

this register is loaded into the interrupt vector base address register, INT\_VECTOR\_BASE.

Units: bytes  
 Default value: 0  
 Minimum value: 0  
 Maximum value: 4194303

Option: -intvbase\_preset\_s  
 Description: This sets the secure interrupt vector base configuration register, VECBASE\_AC\_BUILD. The vector base address is aligned to a 1KB boundary, so the required address value should be divided by 1K (i.e. do not include the lower 10 bits). On reset, this register is loaded into the interrupt vector base address register, INT\_VECTOR\_BASE\_S. This is effective only when 2+2 mode is enabled.

Units: bytes  
 Default value: 0  
 Minimum value: 0  
 Maximum value: 4194303

Option: -rgf\_impl  
 Description: This defines whether the register file is implemented using flip-flops, or with a hard macro.

Units:  
 Default value: flip\_flops  
 Possible values: flip\_flops, macro

Option: -rgf\_num\_regs  
 Description: This defines the size (in 32b register) of the processor register file.

Units: bytes  
 Default value: 16  
 Possible values: 16, 32

Option: -rgf\_wr\_ports  
 Description: This defines the number of write ports on the register file.

Units:  
 Default value: 1  
 Possible values: 1, 2

Option: -rgf\_num\_banks  
 Description: Dual register banks are useful if Fast IRQ has been configured, but may be selected even if not.

Units:  
 Default value: 1  
 Possible values: 1, 2

Option: -rgf\_banked\_REGS  
 Description: This selects the number of registers that are replicated in the second register-file bank.

Units:  
 Default value: 32  
 Possible values: 4, 8, 16, 32

Option: -turbo\_boost

Description: This enables the Turbo Boost synthesis option. By enabling this option, the achievable clock frequency is increased, but at the cost of an additional cycle latency on branch instructions.

Units:

Default value: false

Possible values: true, false

Option: -infer\_alu\_adder

Description: infer: datapath is described as behavioral code: A + B  
instantiate: datapath is instantiated as a detailed multi-stage code of a carry-lookahead adder. It is generally preferable to use the infer option and add directives for your target synthesizer.

Units:

Default value: instantiate

Possible values: instantiate, infer

Option: -infer\_mpy\_wtree

Description: infer: datapath is described as behavioral code: A \* B (applies to only wlh3, wlh4 and wlh5 designs)  
instantiate: datapath is instantiated as a detailed multi-stage code of a Wallace Tree multiplier It is generally preferable to use the infer option and add directives for your target synthesizer.

Units:

Default value: instantiate

Possible values: instantiate, infer

Option: -scantest\_ram\_bypass\_mux

Description: This mux is used to make logic trapped between flops and memory (aka shadow logic) to be covered by scantest without requiring advanced sequential ATPG on the memory to be applied. Will add delay to functional access time

Units:

Default value: false

Possible values: true, false

Option: -logic\_bist

Description: This option will OR LBIST\_EN with test\_mode

Units:

Default value: false

Possible values: true, false

Option: -power\_domains

Description: Adds three separate power domains to the core, and propagates power-gate control signals to the top level of the core. Also generates UPF constraints and commands in the low-power scripts

Units:

Default value: false

Possible values: true, false

Option: -dvfs

Description: Adds logic to the core to allow dynamic controlling of voltage and frequency and propagates the associated control signals to the top level of core

Units:

Default value: false

Possible values: true, false

Option: -mem\_bus\_option

Description: The core supports two bus protocols for accessing external memory: AHB & AHB-Lite. AHB-Lite-single means instruction fetch and data access share a single AHB-Lite port. AHB-Lite-dual means separate AHB-Lite port for each initiator if present.

Units:

Default value: AHB-Lite-dual

Possible values: AHB, AHB-Lite-single, AHB-Lite-dual

Option: -mem\_bus\_reg\_interface

Description: Specifies whether the memory bus interface is registered.

Units:

Default value: false

Possible values: true, false

Option: -dmi\_burst\_option

Description: This will enable high-throughput burst support on the DMI slave interfaces. By enabling this option, the peak DMI read throughput goes from 1 word per 3 cycles to N words per N+2 cycles, in which N is the AHB burst length. DMI write throughput goes from 1 word per 3 cycles to 1 word per cycle.

Units:

Default value: false

Possible values: true, false

Option: -has\_dmp\_peripheral

Description: This option enables the redirection of load/store accesses to one segment (1/16) of the addressable space to a dedicated peripheral bus. This offers high system integration and reduces overall system cost.

Units:

Default value: false

Possible values: true, false

Option: -per\_bus\_option

Description: The core supports one bus protocol for accessing the peripheral space, when enabled: AHB-Lite.

Units:

Default value: AHB-Lite

Possible values: AHB-Lite, APB

Option: -per\_bus\_reg\_interface

Description: Specifies whether the peripheral bus interface is registered.

Units:

Default value: false

Possible values: true, false

Option: -clock\_gating

Description: This enables the insertion of architectural clock gate elements in the design. By enabling this option, the clocks to various parts of the design will be disabled when the logic they drive is not in use to save power.

Units:

Default value: true

Possible values: true, false

Option: -byte\_parity

Description: If parity protection on the CCMs is configured, this option is used to enable parity protection on a per-byte basis. Otherwise, parity will be per word basis

Units:

Default value: false

Possible values: true, false

Option: -prot\_pipelined

Description: Check the box if CCM memories are configured for ECC, and you want single-bit errors to be corrected, written back to memory, and re-fetched. When unchecked, single bit errors are corrected when read from memory, but the offending memory location itself is not corrected with a writeback, no influence on Cache protection

Units:

Default value: true

Possible values: true, false

Option: -cct\_test\_ena

Description: When ECC is configured, this option enables automatic generation of error conditions in relevant testbench memories to exercise error detection and correction features

Units:

Default value: false

Possible values: true, false

---

## D.4 COMPONENT: com.arc.hardware.Calibration\_NVM.1\_0

Option: -cal\_store\_size

Description: This defines the size of calibration nvm component (in 32-bit word).

Units: bytes

Default value: 64

Possible values: 4, 8, 16, 32, 64

Option: -cal\_store\_version

Description: user version of calibration store memory

Units: bytes

Default value: 0

Minimum value: 0

Maximum value: 255

---

## D.5 COMPONENT: com.arc.hardware.CPU\_isle.1\_0

Option: -unique\_name

Description: verilog module modifier prefix

Units:

Default value:  
 Minimum value:  
 Maximum value:

Option: -arc\_num  
 Description: The processor number as read back in the ARCNUM field of the IDENTITY register.  
 Units:  
 Default value: 1  
 Minimum value: 0  
 Maximum value: 255

Option: -instances  
 Description:  
 The number of instantiations of this core.

Units:  
 Default value: 1  
 Minimum value: 1  
 Maximum value: 128

Option: -cpu\_floorplan  
 Description: Floorplan giving relative placement of the RAMs for the given configuration of ARCv2HS or ARCv2EM in this CPUisle  
 Units:  
 Default value: create  
 Possible values: none, create, User, HS34\_base, HS36\_base, HS38\_base, HS36\_cc\_base, HS38\_cc\_base, EV64\_full\_alb\_cpu\_top, EV64\_base\_alb\_cpu\_top, EV64\_hybrid\_scalar\_alb\_cpu\_top, HS38x2\_SMP\_SLC\_full\_alb\_cpu\_top, HS38x4\_SMP\_SLC\_full\_alb\_cpu\_top, em4\_rtos, em4\_sensor, em6\_gp, em4\_ecc, em4, em6, em9d\_yccm, em9d\_xyccm, em11d\_yccm, em4\_parity

Option: -usercpufloorplan\_path  
 Description: Pathname of user floorplan for the CPU when using a hierarchical implementation  
 Units:  
 Default value:  
 Minimum value:  
 Maximum value:

Option: -pin\_location\_constraints\_file  
 Description: Pathname+filename of the physical pin location constraints file or just "side1" (all pins on l.h.s) or "side2" (pins on top only) or "side3" (pins on r.h.s. only) or "side4" (pins on bottom only) to get a template file generated  
 Units:  
 Default value:  
 Minimum value:  
 Maximum value:

---

## D.6 COMPONENT: com.arc.hardware.Data\_Cache.1\_0

Option: -dc\_size  
Description: This defines the total size of the Data Cache in bytes.  
Units: bytes  
Default value: 2048  
Possible values: 2048, 4096, 8192, 16384, 32768

Option: -dc\_ways  
Description: This defines the number of cache ways.  
Units: bytes  
Default value: 1  
Possible values: 1, 2, 4

Option: -dc\_bsize  
Description: This defines the cache line length in bytes.  
Units: bytes  
Default value: 16  
Possible values: 16, 32, 64

Option: -dc\_feature\_level  
Description: Feature Level, indicates locking and debug feature level 00 = Basic cache, with no locking or debug features 01 = Lock and flush features supported 10 = Lock, flush and advanced debug features supported 11 = Reserved  
Units: bytes  
Default value: 0  
Possible values: 0, 1, 2

Option: -dc\_uncached\_region  
Description: Enable an uncached region defined by aux reg  
Units:  
Default value: false  
Possible values: true, false

Option: -dc\_prot  
Description: Specifies the type of protection built for DCACHE.  
Units:  
Default value: None  
Possible values: None, ECC, Parity

Option: -dc\_prot\_level  
Description: Specifies the level of protection.  
Units:  
Default value: Data\_Only  
Possible values: Data\_Only, Data\_Addr

Option: -dc\_prot\_exceptions  
Description: Builds exception generation hardware for uncorrectable (fatal) errors detected on DCACHE.  
Units:  
Default value: true  
Possible values: true, false

---

## D.7 COMPONENT: com.arc.hardware.DCCM.1\_0

```
=====
Option:           -dccm_size
Description:     This defines the size of the Data Closely Coupled Memory (DCCM) in
bytes
Units:          bytes
Default value:  512
Possible values: 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144,
524288, 1048576, 2097152

Option:           -dccm_base
Description:     Sets the initial memory region assignment for DCCM
Units:          bytes
Default value:  8
Possible values: 8, 9, 10, 11, 12, 13, 14, 15

Option:           -dccm_interleave
Description:     Split DCCM into even/odd memory banks.
Units:
Default value:  false
Possible values: true, false

Option:           -dccm_prot
Description:     Specifies the type of protection built for the DCCM.
Units:
Default value:  None
Possible values: None, ECC, Parity

Option:           -dccm_prot_level
Description:     Specifies the level protection.
Units:
Default value:  Data_Only
Possible values: Data_Only, Data_Addr

Option:           -dccm_prot_exceptions
Description:     When the core is configured with ECC or Parity, cause exception
generation hardware to be created for uncorrectable errors detected on the DCCM
Units:
Default value:  true
Possible values: true, false

Option:           -dccm_sec_lvl
Description:     Specifies the level of secure DCCM.
Units:
Default value:  Non_Secure
Possible values: Non_Secure, Secure, Halves

Option:           -dccm_dmi
```

Description: This enables external access through a DMI (direct memory interface) port.  
 Units:  
 Default value: false  
 Possible values: true, false

---

## D.8 COMPONENT: com.arc.hardware.Debug\_Interface.1\_0

Option: -secure\_debug  
 Description: This enables secure debug feature  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -scdbg\_aux\_unlk  
 Description: Example auxiliary based unlock is enabled.  
 Units:  
 Default value: false  
 Possible values: true, false

---

## D.9 COMPONENT: com.arc.hardware.DMA\_Controller.1\_0

Option: -dmac\_channels  
 Description: This option specifies the number of DMA channels implemented in the DMA controller  
 Units: bytes  
 Default value: 1  
 Possible values: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

Option: -dmac\_fifo\_depth  
 Description: This option specifies the DMA transfer FIFO depth in 32b words.  
 Units: bytes  
 Default value: 1  
 Possible values: 1, 2, 4

Option: -dmac\_int\_config  
 Description: None: the DMA controller cannot raise an interrupt  
 Single-External: single done and single error interrupt signal for all DMA channels, and the interrupt signals are routed to a port at the top of the EM logical hierarchy  
 Multiple-External: each DMA channel can be configured to raise separate (per-channel) done and error interrupts, and the interrupt signals are routed to ports at the top of the EM logical hierarchy  
 Single-Internal: single done and single error interrupt signals for all DMA channels, and the interrupt signals are internal to the EM core  
 Multiple-Internal: each DMA channel can be configured to raise separate (per-channel) done and error interrupts, and the interrupt signals are internal to the EM core  
 Units:

Default value: Single-Internal  
 Possible values: None, Single-Internal, Multiple-Internal, Single-External, Multiple-External

Option: -dmac\_registers  
 Description: This option defines the number of DMA channels with their registers located in auxiliary space.  
 Units: bytes  
 Default value: 0  
 Minimum value: 0  
 Maximum value: 16

Option: -dmac\_mem\_if  
 Description: This option specifies whether the DMA controller system memory interface is integrated into the existing EM system memory interfaces or has its own interface.  
 Units:  
 Default value: integrated  
 Possible values: integrated, separate

Option: -dmac\_per\_if  
 Description: Internal vs DW peripheral interface. Specify (in hex) which channels have the DW interface, where bit 0 corresponds to DMA channel 0, bit 1 for DMA channel 1, etc.  
 Example: 4 channel DMA controller where -dmac\_per\_if is set to 0x9 = DMA Channels 0 and 3 configured with the DW req interface, DMA Channels 1 and 2 configured with the internal req interface.  
 Units: bytes  
 Default value: 0  
 Minimum value: 0  
 Maximum value: 65535

---

## D.10 COMPONENT: com.arc.hardware.DSP.1\_0

Option: -dsp\_complex  
 Description: Enable/disable support for single cycle 16b+16b complex instructions and butterfly operations, else 2-cycle complex instructions only without butterfly support  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -dsp\_itu  
 Description: Enable/disable support for ITU bit-accurate 1 bit fractional shift before accumulation, else 1-bit fractional shift result after accumulation only  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -dsp\_divsqrt

Description: Enable/disable support for divide and square root operations: DIV(U), REM(U), SQRT  
 Units:  
 Default value: radix2  
 Possible values: none, radix2, radix4

Option: -dsp\_accshift  
 Description: Select support for accumulator shift operations: no supported, limited shift support only or full shift support and convergent rounding  
 Units:  
 Default value: limited  
 Possible values: limited, full

Option: -dsp\_impl  
 Description: The datapath components may be inferred from Verilog for better area or optimized using carry-save components for better timing  
 Units:  
 Default value: optimized  
 Possible values: inferred, optimized

---

## D.11 COMPONENT: com.arc.hardware.Floating\_point\_unit.1\_0

Option: -fpu\_dp\_assist  
 Description: This enables double-precision acceleration instructions.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -fpu\_fma\_option  
 Description: This enables the fused multiply-add & multiply-subtract instructions.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -fpu\_mas\_cycles  
 Description: Make mul/add/sub multicycle to achieve a higher clock speed.  
 Units: bytes  
 Default value: 1  
 Possible values: 1, 2

Option: -fpu\_div\_option  
 Description: This enables divide & square-root acceleration  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -fpu\_div\_cycles  
 Description: Controls div/sqrt implementation.  
 Units: bytes

Default value: 17  
 Possible values: 1, 17

---

## D.12 COMPONENT: com.arc.hardware.ICCM0\_1\_0

Option: -iccm0\_size  
 Description: This defines the size of ICCM0 in bytes. This ICCM has 0 wait states.  
 Units: bytes  
 Default value: 512  
 Possible values: 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152

Option: -iccm0\_base  
 Description: Sets the initial memory region assignment for ICCM0  
 Units: bytes  
 Default value: 0  
 Possible values: 0, 1, 2, 3, 4, 5, 6, 7

Option: -iccm0\_wide  
 Description: Creates ICCM0 as 64b memory to reduce accesses.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -iccm0\_prot  
 Description: Specifies the type of protection built for ICCM0.  
 Units:  
 Default value: None  
 Possible values: None, ECC, Parity

Option: -iccm0\_prot\_level  
 Description: Specifies the level of protection.  
 Units:  
 Default value: Data\_Only  
 Possible values: Data\_Only, Data\_Addr

Option: -iccm0\_prot\_exceptions  
 Description: When the core is configured with ECC or Parity, cause exception generation hardware to be created for uncorrectable errors detected on the ICCM0  
 Units:  
 Default value: true  
 Possible values: true, false

Option: -iccm0\_sec\_lvl  
 Description: Specifies the level of secure ICCM0.  
 Units:  
 Default value: Non\_Secure  
 Possible values: Non\_Secure, Secure, Halves

Option: -iccm0\_dmi

Description: This enables external access through a DMI (direct memory interface) port.  
Units:  
Default value: false  
Possible values: true, false

=====

## D.13 COMPONENT: com.arc.hardware.ICCM1.1\_0

Option: -iccm1\_size  
Description: This defines the size of ICCM1 in bytes. This ICCM has 1 wait states.  
Units: bytes  
Default value: 512  
Possible values: 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152

Option: -iccm1\_base  
Description: Sets the initial memory region assignment for ICCM1  
Units: bytes  
Default value: 0  
Possible values: 0, 1, 2, 3, 4, 5, 6, 7

Option: -iccm1\_banking  
Description: With dual memory banks, the ICCM has better cycle performance.  
Units:  
Default value: false  
Possible values: true, false

Option: -iccm1\_sec\_lvl  
Description: Specifies the level of secure ICCM1.  
Units:  
Default value: Non\_Secure  
Possible values: Non\_Secure, Secure, Halves

Option: -iccm1\_dmi  
Description: This enables external access through a DMI (direct memory interface) port.  
Units:  
Default value: false  
Possible values: true, false

=====

## D.14 COMPONENT: com.arc.hardware.Instruction\_Cache.1\_0

Option: -ic\_size  
Description: This defines the total size of the instruction cache in bytes.  
Units: bytes  
Default value: 2048  
Possible values: 2048, 4096, 8192, 16384, 32768

Option: -ic\_ways  
 Description: This defines the number of cache ways  
 Units: bytes  
 Default value: 1  
 Possible values: 1, 2, 4

Option: -ic\_bsize  
 Description: This defines the cache line length in bytes.  
 Units: bytes  
 Default value: 16  
 Possible values: 16, 32, 64

Option: -ic\_disable\_on\_reset  
 Description: The instruction cache may be enabled immediately after reset, depending on this option. If this option is enabled, the last cache operation is set to failed, and the direct cache-RAM access is enabled. Furthermore, the instruction cache is invalidated all cache lines are invalidated and unlocked, and the tag RAM is cleared.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -ic\_feature\_level  
 Description: This defines the feature level of the cache.  
 Units: bytes  
 Default value: 0  
 Possible values: 0, 1, 2

Option: -ic\_pwr\_opt\_level  
 Description: This selects power-optimization options in the micro-architecture of the instruction cache.  
 Units: bytes  
 Default value: 1  
 Possible values: 0, 1

Option: -ic\_prot  
 Description: Specifies the type of protection built for ICACHE.  
 Units:  
 Default value: None  
 Possible values: None, ECC, Parity

Option: -ic\_prot\_level  
 Description: Specifies the level of protection.  
 Units:  
 Default value: Data\_Only  
 Possible values: Data\_Only, Data\_Addr

Option: -ic\_prot\_exceptions  
 Description: Builds exception generation hardware for uncorrectable (fatal) errors detected on ICACHE.  
 Units:  
 Default value: true

Possible values: true, false

## D.15 COMPONENT: com.arc.hardware.Instruction\_Fetch\_Queue.1\_0

Option: -ifqueue\_size

Description: This defines the number of entires in the Instruction Fetch Queue.

Units: words

Default value: 1

Possible values: 1, 4, 8

Option: -ifqueue\_burst\_size

Description: This sets the burst size for bus data transfers (in 32-bit words).

It cannot exceed the number of entries.

Units: words

Default value: 1

Possible values: 1, 2, 4, 8

## D.16 COMPONENT: com.arc.hardware.Interrupt\_Controller.1\_0

Option: -number\_of\_interrupts

Description: This is the total number of interrupts available to the core. Some interrupts are allocated statically to a specific interrupt line (for example, timer interrupts). For more information on Interrupt and register-file options, see DesignWare ARCv2 ISA Programmers Reference Manual.

Units: bytes

Default value: 32

Minimum value: 1

Maximum value: 240

Option: -number\_of\_levels

Description: Priority levels in the interrupt controller.

Units: bytes

Default value: 2

Possible values: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

Option: -external\_interrupts

Description: This is the total number of interrupt pins available for external system components. This parameter must be less than the total number of interrupts.

Units: bytes

Default value: 0

Minimum value: 0

Maximum value: 240

Option: -firq\_option

Description: This enables the fast-interrupts option, (priority level 0 interrupts), which uses an alternate register bank (if configured) instead of saving the context to memory.

Units:  
 Default value: false  
 Possible values: true, false

---

## D.17 COMPONENT: com.arc.hardware.Key\_NVM.1\_0

Option: -key\_store\_size  
 Description: This defines the size of key nvm component (in 32-bit word).  
 Units: bytes  
 Default value: 256  
 Possible values: 4, 8, 16, 32, 64, 128, 256

Option: -key\_store\_version  
 Description: user version of key store memory  
 Units: bytes  
 Default value: 0  
 Minimum value: 0  
 Maximum value: 255

---

## D.18 COMPONENT: com.arc.hardware.Lockstep\_Comparator.1\_0

Option: -ls\_delay  
 Description: Specifies the number of delay stages for Time Diversity  
 Units: bytes  
 Default value: 2  
 Possible values: 0, 1, 2

Option: -ls\_parity  
 Description: Enable or Disable Parity protection on delay buffers  
 Units: bytes  
 Default value: 1  
 Possible values: 0, 1

Option: -ls\_ainp\_sync\_stages  
 Description: Specifies the number of Synchronizer stages for Asynchronous Inputs  
 Units: bytes  
 Default value: 2  
 Possible values: 1, 2

Option: -ls\_eirq\_sync\_stages  
 Description: Specifies the number of Synchronizer stages for Asynchronous Interrupts  
 Units: bytes  
 Default value: 2  
 Possible values: 1, 2

Option: -srams\_signal\_prefix

Description: Specifies the signal prefix for CCM SRAMs  
Units:  
Default value: lssrams\_  
Minimum value:  
Maximum value:  
  
Option: -srams\_config  
Description: Single: Shared CCM, double: Separate CCM  
Units:  
Default value: double  
Possible values: single, double

=====

## D.19 COMPONENT: com.arc.hardware.Memory\_Protection\_Unit.1\_0

Option: -mpu\_num\_regions  
Description: Number of configured memory regions.  
Units: bytes  
Default value: 8  
Possible values: 1, 2, 4, 8, 16  
  
Option: -mpu\_32b  
Description: Set the minimal region size to be 32 byte instead of 2KB.  
Units:  
Default value: false  
Possible values: true, false  
  
Option: -mpu\_sid\_option  
Description: It will enable SID support in Secure Shield  
Units:  
Default value: false  
Possible values: true, false

=====

## D.20 COMPONENT: com.arc.hardware.Performance\_Monitor.1\_0

Option: -pct\_counters  
Description: Number of counters for performance monitoring.  
Units: bytes  
Default value: 8  
Possible values: 4, 8, 16, 32

=====

## D.21 COMPONENT: com.arc.hardware.Program\_NVM.1\_0

Option: -nv\_iccm\_size  
Description: This defines the size of nvm program storage (in bytes).

Units: bytes  
 Default value: 2048  
 Possible values: 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144

Option: -nv\_iccm\_ifq  
 Description: Added IFQ to design for nvm program storage  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -nv\_iccm\_base  
 Description: Sets the initial memory region assignment for NVM program storage  
 Units: bytes  
 Default value: 7  
 Possible values: 0, 1, 2, 3, 4, 5, 6, 7

Option: -nvm\_ecc\_present  
 Description: Use NVM ECC internal function or not  
 Units:  
 Default value: false  
 Possible values: true, false

---

## D.22 COMPONENT: com.arc.hardware.Real\_time\_trace\_producer.1\_0

Option: -rtt\_feature\_level  
 Description: 'small' means that program trace only is available. 'medium' adds data trace. 'full' adds core and aux register trace.  
 Units:  
 Default value: small  
 Possible values: small, medium, full

---

## D.23 COMPONENT: com.arc.hardware.Secure\_pipeline\_features.1\_0

Option: -sec\_scramble  
 Description: Enable insertion of scrambling modules on the address & data lines for embedded SRAMs and external memory interfaces.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -sec\_encrypt\_i  
 Description: Add support for user defined instruction encryption  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -sec\_encrypt\_d

```
Description:      Add support for user defined data encryption
Units:
Default value:  false
Possible values: true, false

Option:          -err_prot_option
Description:     Select the ECC mode for CCM ECC protection: false: detect & correct
or true: detect & fail
Units:
Default value:  false
Possible values: true, false

Option:          -pipe_prot_option
Description:     Select to enable PC & Register File error protection
Units:
Default value:  false
Possible values: true, false

Option:          -sec_modes_option
Description:     Enable secure shield 2+2 mode
Units:
Default value:  false
Possible values: true, false
```

---

## D.24 COMPONENT: com.arc.hardware.Secure\_Timer\_0\_1\_0

```
Option:          -stimer_0_int_level
Description:    This sets the interrupt level (and implicitly the priority: level 0
is highest) of secure timer 0.
Units:          bytes
Default value:  1
Possible values: 0, 1, 2, 3, 4, 5, 6, 7
```

---

## D.25 COMPONENT: com.arc.hardware.Secure\_Timer\_1\_1\_0

```
Option:          -stimer_1_int_level
Description:    This sets the interrupt level (and implicitly the priority: level 0
is highest) of secure timer 1.
Units:          bytes
Default value:  0
Possible values: 0, 1, 2, 3, 4, 5, 6, 7
```

---

## D.26 COMPONENT: com.arc.hardware.SmaRT.1\_0

```

Option:          -smart_stack_entries
Description:    This specifies the number of entries in the trace buffer.
Units:          bytes
Default value:  8
Possible values: 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096

Option:          -smart_implementation
Description:    Flip-flop = FF-based design. Memory = memory-based design (provides
better density for larger trace buffers).
Units:
Default value: flip-flop
Possible values: flip-flop, memory
=====
```

## D.27 COMPONENT: com.arc.hardware.System.1\_0

```

Option:          -testbench
Description:
Only the rascal testbench is supported, and is required by ARCTest.

Units:          no units
Default value: rascal
Possible values: standalone, rascal

Option:          -synthesislevel
Description:
Sets the top level module name for synthesis.

If not using core_sys: for single-core designs, cpu_isle is used; for multicore designs,
archipelago is used.

Units:
Default value: cpu_isle/archipelago
Possible values: core_sys, cpu_isle/archipelago

Option:          -gatesim
Description:    When selected the gate level sim test code and scripts would be
installed to run ARCGatesim
Units:
Default value: true
Possible values: true, false

Option:          -library_name
Description:    The name for your HDL library
Units:          no units
Default value: user
Minimum value: 0
Maximum value: 10

Option:          -simulator
Description:    The name of the simulator you wish to use
```

Units: no units

Default value: vcs

Possible values: none, modelsim, vcs, nc

Option: -sim64

Description: When selected, the 64-bit version of the simulator is used. Be sure you have the 64-bit-capable simulator installed and \$ARCHITECT\_ROOT/lib/linux\_x86\_64/ added to your LD\_LIBRARY\_PATH.

The setting of this option affects the content of the generated makefile\_interface\_\*\_verilog, where \* is the simulator name.

Units:

Default value: false

Possible values: true, false

Option: -verilog\_2001

Description: Enable Verilog 2001 file-io syntax (if false: use pli)

Units:

Default value: true

Possible values: true, false

=====

## D.28 COMPONENT: com.arc.hardware.Timer\_0.1\_0

Option: -timer\_0\_int\_level

Description: This sets the interrupt level (and implicitly the priority: level 0 is highest) of timer 0.

Units: bytes

Default value: 1

Possible values: 0, 1, 2, 3, 4, 5, 6, 7

=====

## D.29 COMPONENT: com.arc.hardware.Timer\_1.1\_0

Option: -timer\_1\_int\_level

Description: This sets the interrupt level (and implicitly the priority: level 0 is highest) of timer 1.

Units: bytes

Default value: 0

Possible values: 0, 1, 2, 3, 4, 5, 6, 7

=====

## D.30 COMPONENT: com.arc.hardware.Watchdog\_Timer.1\_0

Option: -watchdog\_size

Description: Specifies the bit width of the internal counter used within the timer.

Units: bytes

Default value: 16  
Possible values: 16, 32

Option: -watchdog\_clk  
Description: Specifies whether the timer should be driven from a separate clock.  
Units:  
Default value: false  
Possible values: true, false

=====

COMPONENT: com.arc.hardware.XY.1\_0

Option: -xy\_config  
Description: XY memory configuration:  
One memory: DCCM only.  
Two memories: DCCM + Y.  
Three memories: DCCM + X + Y.  
Units:  
Default value: dccm\_x\_y  
Possible values: dccm, dccm\_y, dccm\_x\_y

Option: -xy\_size  
Description: Size of X and Y memories if included.  
X and Y memories both have the same configured size.  
Units: bytes  
Default value: 4096  
Possible values: 4096, 8192, 16384, 32768, 65536

Option: -xy\_interleave  
Description: Split XY memories into odd/even instances to enable single cycle unaligned access.  
Units:  
Default value: true  
Possible values: true, false

Option: -xy\_x\_base  
Description: Base region for X memory. All accesses to this region will initiate a transfer on the X memory.  
Units: bytes  
Default value: 12  
Possible values: 8, 9, 10, 11, 12, 13, 14, 15

Option: -xy\_y\_base  
Description: Base region for Y memory. All accesses to this region will initiate a transfer on the Y memory.  
Units: bytes  
Default value: 14  
Possible values: 8, 9, 10, 11, 12, 13, 14, 15

=====

## D.31 COMPONENT: com.arc.hardware.XY\_DMA\_Controller.1\_0

Option: -dmac\_fifo\_depth

Description: This option specifies the DMA transfer FIFO depth in 32b words.

Units: bytes

Default value: 1

Possible values: 1, 2, 4

Option: -dmac\_int\_config

Description: None: the DMA controller cannot raise an interrupt

Single-External: single done and single error interrupt signal for all DMA channels, and the interrupt signals are routed to a port at the top of the EM logical hierarchy

Multiple-External: each DMA channel can be configured to raise separate (per-channel) done and error interrupts, and the interrupt signals are routed to ports at the top of the EM logical hierarchy

Single-Internal: single done and single error interrupt signals for all DMA channels, and the interrupt signals are internal to the EM core

Multiple-Internal: each DMA channel can be configured to raise separate (per-channel) done and error interrupts, and the interrupt signals are internal to the EM core

Units:

Default value: Single-Internal

Possible values: None, Single-Internal

Option: -dmac\_mem\_if

Description: This option specifies whether the DMA controller system memory interface is integrated into the existing EM system memory interfaces or has its own interface.

Units:

Default value: integrated

Possible values: integrated, separate

=====

## D.32 COMPONENT: com.arc.hardware.ARConnect.1\_0

Option: -mcip\_def\_div2ref

Description: This specifies the clock division factor at reset. This option is irrelevant to any RTL featuresIt is only used for ARCv2MSS clock controller to generate ARConnect clock in ARChitect simulation environment, and the value N means ARConnect is running at (1/N) x ref\_clk.

Units: bytes

Default value: 1

Minimum value: 1

Maximum value: 255

Option: -mcip\_has\_cdc

Description: This specifies whether Clock-Domain-Crossing logics implemented in all the interfaces between cores and ARConnect. If this option is configured, the ARConnect can be put in completely-independent clock domain asynchronous to connected cores.

**Units:**  
 Default value: false  
 Possible values: true, false

**Option:** -mcip\_has\_intrpt  
**Description:** This specifies whether the Inter-core Interrupt Unit exists. NOTE: in single-core configuration the ICI can only be used to generate interrupt to external system, other inter-core interrupt functionalities are pointless.

**Units:**  
 Default value: true  
 Possible values: true, false

**Option:** -mcip\_has\_sema  
**Description:** This specifies whether the Inter-core Semaphore Unit exists. NOTE: in single-core configuration the inter-core semaphores functionalities are pointless.

**Units:**  
 Default value: true  
 Possible values: true, false

**Option:** -mcip\_sema\_num  
**Description:** This specifies the number of semaphores in the Inter-core Semaphores Unit

**Units:** bytes  
 Default value: 16  
 Possible values: 8, 16, 32

**Option:** -mcip\_has\_msg\_sram  
**Description:** This specifies whether the Inter-core Message Unit exists. NOTE: in single-core configuration the inter-core message functionalities are pointless.

**Units:**  
 Default value: false  
 Possible values: true, false

**Option:** -mcip\_msg\_sram\_size  
**Description:** This specifies the bytes of SRAM in the Inter-core Message Unit

**Units:** bytes  
 Default value: 512  
 Possible values: 128, 256, 512, 1024, 2048, 4096

**Option:** -mcip\_msg\_mem\_posedge  
**Description:** True: The access path to message SRAM is 1 clock cycle, and inter-core message passing SRAM are clocked on the positive edge of the clock; False: The access path to message SRAM is 1.5 cycles, inter-core message passing SRAM are clocked on the negative edge of the clock. Note: The 1.5 cycles path use clock negetive edge for SRAM, but can acheive higher frequency. No performance difference caused by the value of this option

**Units:**  
 Default value: false  
 Possible values: true, false

**Option:** -mcip\_msg\_ecc

Description: This specifies the ECC mode of message SRAM. none = No checking; parity = Parity only; SECDED = single-error correction and double-error detection (SECDED)

Units:

Default value: none

Possible values: none, parity, SECDED

Option: -mcip\_has\_debug

Description: This specifies whether the Inter-core Debug Unit exists. NOTE: in single-core configuration the inter-core debug functionalities are pointless.

Units:

Default value: true

Possible values: true, false

Option: -mcip\_has\_grtc

Description: This specifies whether the Global Real-Time Counter Unit exists.

Units:

Default value: true

Possible values: true, false

Option: -mcip\_has\_pmu

Description: Specifies whether ARConnect Power Management Unit Exists. If disabled, external PMU should be used. If enabled, internal ARConnect PMU is used. This option takes effect only if Power Domains are enabled

Units:

Default value: false

Possible values: true, false

Option: -mcip\_power\_domains

Description: Enables Power Domains so that the ARConnect logic and memories are split between Always-On, PD1 and PD2 domains

Units:

Default value: false

Possible values: true, false

=====

### D.33 COMPONENT: com.arc.hardware.AR Cv2MSS.BusFabric.1\_0

Option: -alb\_mss\_fab\_def\_div2ref

Description: This specifies the clock division factor at reset. It is used for mss clock controller to generate mss fabric clock, and the value N means mss fabric is running at (1/N) x ref\_clk.

Units:

Default value: 2

Minimum value: 1

Maximum value: 255

Option: -alb\_mss\_fab\_lat

Description: This specifies the maximum latency in the master latency units.

Units:

Default value: 0

Possible values: 0, 8, 16, 32, 64, 128, 256

Option: -alb\_mss\_fab\_def\_lat

Description: This specifies the latency after reset for the master latency units.

Units:

Default value: 0

Minimum value: 0

Maximum value: 256

Option: -alb\_mss\_ccm\_base

Description: This specifies the base address at which the ICCM and DCCM DMIs will be placed in the memory map. The address should be divided by 4KB i.e. do not specify the lower 12 bits of the address.

Units:

Default value: 262144

Minimum value: 0

Maximum value: 1048320

=====

## D.34 COMPONENT: com.arc.hardware.ARcv2MSS.ClkCtrl.1\_0

Option: -alb\_mss\_clkctrl\_base\_addr

Description: This specifies the clock controller base address in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address.

Units:

Default value: 786432

Minimum value: 0

Maximum value: 1048575

Option: -alb\_mss\_clkctrl\_bypass\_mode

Description: If true then all clock dividers/gaters in the clock controller are bypassed, clock ratio is not supported and the division options/registers are overridden

Units:

Default value: false

Possible values: true, false

=====

## D.35 COMPONENT: com.arc.hardware.ARcv2MSS.DummyMST.1\_0

Option: -alb\_mss\_dummy\_master\_num

Description: Number of dummy masters

Units:

Default value: 1

Possible values: 1, 2, 3, 4

Option: -alb\_mss\_dummy\_master1\_pref

Description: This specifies the unique prefix of dummy master 1.

Units:  
Default value: dmst1\_  
Minimum value:  
Maximum value:

Option: -alb\_mss\_dummy\_master1\_prot  
Description: This specifies the protocol of the dummy master 1.  
Units:  
Default value: AXI  
Possible values: AXI, AHB\_Lite, BVCI, AHB5

Option: -alb\_mss\_dummy\_master1\_dw  
Description: This specifies the data width of dummy master 1.  
Units:  
Default value: 32  
Possible values: 32, 64

Option: -alb\_mss\_dummy\_master2\_pref  
Description: This specifies the unique prefix of dummy master 2.  
Units:  
Default value: dmst2\_  
Minimum value:  
Maximum value:

Option: -alb\_mss\_dummy\_master2\_prot  
Description: This specifies the protocol of the dummy master 2.  
Units:  
Default value: AHB\_Lite  
Possible values: AXI, AHB\_Lite, BVCI, AHB5

Option: -alb\_mss\_dummy\_master2\_dw  
Description: This specifies the data width of dummy master 2.  
Units:  
Default value: 32  
Possible values: 32, 64

Option: -alb\_mss\_dummy\_master3\_pref  
Description: This specifies the unique prefix of dummy master 3.  
Units:  
Default value: dmst3\_  
Minimum value:  
Maximum value:

Option: -alb\_mss\_dummy\_master3\_prot  
Description: This specifies the protocol of the dummy master 3.  
Units:  
Default value: AHB5  
Possible values: AXI, AHB\_Lite, BVCI, AHB5

Option: -alb\_mss\_dummy\_master3\_dw  
Description: This specifies the data width of dummy master 3.  
Units:  
Default value: 32

Possible values: 32, 64

Option: -alb\_mss\_dummy\_master4\_pref  
 Description: This specifies the unique prefix of dummy master 4.

Units:

Default value: dmst4\_

Minimum value:

Maximum value:

Option: -alb\_mss\_dummy\_master4\_prot  
 Description: This specifies the protocol of the dummy master 4.  
 Units:  
 Default value: BVCI  
 Possible values: AXI, AHB\_Lite, BVCI, AHB5

Option: -alb\_mss\_dummy\_master4\_dw  
 Description: This specifies the data width of dummy master 4.  
 Units:  
 Default value: 32  
 Possible values: 32, 64

=====

## D.36 COMPONENT: com.arc.hardware.AR Cv2MSS.DummySLV.1\_0

Option: -alb\_mss\_dummy\_slave\_num  
 Description: This specifies the number of dummy slaves.  
 Units:  
 Default value: 1  
 Possible values: 1, 2, 3, 4, 5

Option: -alb\_mss\_dummy\_slave1\_base\_addr  
 Description: This specifies the base address of dummy slave 1 in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address. The base address of dummy slave should be aligned with the size of the dummy slave  
 Units:  
 Default value: 786438  
 Minimum value: 0  
 Maximum value: 1048575

Option: -alb\_mss\_dummy\_slave1\_size  
 Description: This specifies the size of dummy slave 1.  
 Units:  
 Default value: 4KB  
 Possible values: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

Option: -alb\_mss\_dummy\_slave1\_pref  
 Description: This specifies the unique prefix of dummy slave 1.  
 Units:  
 Default value: dslv1\_  
 Minimum value:

Maximum value:

Option: -alb\_mss\_dummy\_slave1\_prot  
Description: This specifies the protocol of dummy slave 1.  
Units:  
Default value: APB  
Possible values: AXI, AHB\_Lite, BVCI, APB

Option: -alb\_mss\_dummy\_slave1\_dw  
Description: This specifies the data width of dummy slave 1.  
Units:  
Default value: 32  
Possible values: 32, 64

Option: -alb\_mss\_dummy\_slave2\_base\_addr  
Description: This specifies the base address of dummy slave 2 in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address. The base address of dummy slave should be aligned with the size of the dummy slave  
Units:  
Default value: 786694  
Minimum value: 0  
Maximum value: 1048575

Option: -alb\_mss\_dummy\_slave2\_size  
Description: This specifies the size of dummy slave 2.  
Units:  
Default value: 4KB  
Possible values: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

Option: -alb\_mss\_dummy\_slave2\_pref  
Description: This specifies the unique prefix of dummy slave 2.  
Units:  
Default value: dslv2\_  
Minimum value:  
Maximum value:

Option: -alb\_mss\_dummy\_slave2\_prot  
Description: This specifies the protocol of dummy slave 2.  
Units:  
Default value: AXI  
Possible values: AXI, AHB\_Lite, BVCI, APB

Option: -alb\_mss\_dummy\_slave2\_dw  
Description: This specifies the data width of dummy slave 2.  
Units:  
Default value: 32  
Possible values: 32, 64

Option: -alb\_mss\_dummy\_slave3\_base\_addr  
Description: This specifies the base address of dummy slave 3 in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address. The base address of dummy slave should be aligned with the size of the dummy slave

**Units:**  
**Default value:** 786950  
**Minimum value:** 0  
**Maximum value:** 1048575

**Option:** -alb\_mss\_dummy\_slave3\_size  
**Description:** This specifies the size of dummy slave 3.  
**Units:**  
**Default value:** 4KB  
**Possible values:** 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

**Option:** -alb\_mss\_dummy\_slave3\_pref  
**Description:** This specifies the unique prefix of dummy slave 3.  
**Units:**  
**Default value:** dslv3\_  
**Minimum value:**  
**Maximum value:**

**Option:** -alb\_mss\_dummy\_slave3\_prot  
**Description:** This specifies the protocol of dummy slave 3.  
**Units:**  
**Default value:** AHB\_Lite  
**Possible values:** AXI, AHB\_Lite, BVCI, APB

**Option:** -alb\_mss\_dummy\_slave3\_dw  
**Description:** This specifies the data width of dummy slave 3.  
**Units:**  
**Default value:** 32  
**Possible values:** 32, 64

**Option:** -alb\_mss\_dummy\_slave4\_base\_addr  
**Description:** This specifies the base address of dummy slave 4 in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address. The base address of dummy slave should be aligned with the size of the dummy slave  
**Units:**  
**Default value:** 787206  
**Minimum value:** 0  
**Maximum value:** 1048575

**Option:** -alb\_mss\_dummy\_slave4\_size  
**Description:** This specifies the size of dummy slave 4.  
**Units:**  
**Default value:** 4KB  
**Possible values:** 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

**Option:** -alb\_mss\_dummy\_slave4\_pref  
**Description:** This specifies the unique prefix of dummy slave 4.  
**Units:**  
**Default value:** dslv4\_  
**Minimum value:**  
**Maximum value:**

Option: -alb\_mss\_dummy\_slave4\_prot  
Description: This specifies the protocol of dummy slave 4.  
Units:  
Default value: BVCI  
Possible values: AXI, AHB\_Lite, BVCI, APB

Option: -alb\_mss\_dummy\_slave4\_dw  
Description: This specifies the data width of dummy slave 4.  
Units:  
Default value: 32  
Possible values: 32, 64

Option: -alb\_mss\_dummy\_slave5\_base\_addr  
Description: This specifies the base address of dummy slave 5 in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address. The base address of dummy slave should be aligned with the size of the dummy slave  
Units:  
Default value: 787462  
Minimum value: 0  
Maximum value: 1048575

Option: -alb\_mss\_dummy\_slave5\_size  
Description: This specifies the size of dummy slave 5.  
Units:  
Default value: 4KB  
Possible values: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

Option: -alb\_mss\_dummy\_slave5\_pref  
Description: This specifies the unique prefix of dummy slave 5.  
Units:  
Default value: dslv5\_  
Minimum value:  
Maximum value:

Option: -alb\_mss\_dummy\_slave5\_prot  
Description: This specifies the protocol of dummy slave 5.  
Units:  
Default value: BVCI  
Possible values: AXI, AHB\_Lite, BVCI, APB

Option: -alb\_mss\_dummy\_slave5\_dw  
Description: This specifies the data width of dummy slave 5.  
Units:  
Default value: 32  
Possible values: 32, 64

=====

## D.37 COMPONENT: com.arc.hardware.AR Cv2MSS.Profiler.1\_0

Option: -alb\_mss\_perfctrl\_base\_addr  
 Description: This specifies the profiler base address in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address.  
 Units:  
 Default value: 786433  
 Minimum value: 0  
 Maximum value: 1048575

---

## D.38 COMPONENT: com.arc.hardware.AR Cv2MSS.SnoopTrafficGen.1\_0

Option: -alb\_mss\_snoop\_trafgen\_base\_addr  
 Description: This specifies the snoop traffic generator base address in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address.  
 Units:  
 Default value: 786434  
 Minimum value: 0  
 Maximum value: 1048575

---

## D.39 COMPONENT: com.arc.hardware.AR Cv2MSS.SRAMCtrl.1\_0

Option: -alb\_mss\_mem\_region\_num  
 Description: The number of regions supported in SRAMCtrl component  
 Units:  
 Default value: 1  
 Possible values: 1, 2, 3, 4

Option: -alb\_mss\_mem0\_base\_addr  
 Description: This specifies the base address of memory region 0 in the memory map, divided by 4KB i.e. do no specify the lower 12 bits of the address.  
 Units:  
 Default value: 0  
 Minimum value:  
 Maximum value:

Option: -alb\_mss\_mem0\_size  
 Description: This specifies the memory size of memory region 0.  
 Units:  
 Default value: 1MB  
 Possible values: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

Option: -alb\_mss\_mem0\_attr  
 Description: This specifies the memory access attribute of memory region 0.  
 Units:  
 Default value: Read-Write  
 Possible values: Read-only, Write-only, Read-Write

Option: -alb\_mss\_mem0\_secure

Description: This specifies the memory secure attribute of memory region 0.

Units:

Default value: Non-Secure

Possible values: Non-Secure, Secure

Option: -alb\_mss\_mem0\_lat

Description: This specifies the maximum latency of memory region 0 .

Units:

Default value: 0

Possible values: 0, 8, 16, 32, 64, 128, 256, 512, 1024

Option: -alb\_mss\_mem0\_def\_lat

Description: This specifies the latency after reset of memory region 0 .

Units:

Default value: 0

Minimum value: 0

Maximum value: 1024

Option: -alb\_mss\_mem1\_base\_addr

Description: This specifies the base address of memory region 1 in the memory map, divided by 4KB i.e. do no specify the lower 12 bits of the address.

Units:

Default value: 1048576

Minimum value:

Maximum value:

Option: -alb\_mss\_mem1\_size

Description: This specifies the memory size of memory region 1.

Units:

Default value: 1MB

Possible values: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

Option: -alb\_mss\_mem1\_attr

Description: This specifies the memory access attribute of memory region 1 .

Units:

Default value: Read-Write

Possible values: Read-only, Write-only, Read-Write

Option: -alb\_mss\_mem1\_secure

Description: This specifies the memory secure attribute of memory region 1 .

Units:

Default value: Non-Secure

Possible values: Non-Secure, Secure

Option: -alb\_mss\_mem1\_lat

Description: This specifies the maximum latency of memory region 1 .

Units:

Default value: 0

Possible values: 0, 8, 16, 32, 64, 128, 256, 512, 1024

Option: -alb\_mss\_mem1\_def\_lat

Description: This specifies the latency after reset of memory region 1.

Units:

Default value: 0

Minimum value: 0

Maximum value: 1024

Option: -alb\_mss\_mem2\_base\_addr

Description: This specifies the base address of memory region 2 in the memory map, divided by 4KB i.e. do no specify the lower 12 bits of the address.

Units:

Default value: 1048576

Minimum value:

Maximum value:

Option: -alb\_mss\_mem2\_size

Description: This specifies the memory size of memory region 2.

Units:

Default value: 1MB

Possible values: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

Option: -alb\_mss\_mem2\_attr

Description: This specifies the memory access attribute of memory region 2.

Units:

Default value: Read-Write

Possible values: Read-only, Write-only, Read-Write

Option: -alb\_mss\_mem2\_secure

Description: This specifies the memory secure attribute of memory region 2.

Units:

Default value: Non-Secure

Possible values: Non-Secure, Secure

Option: -alb\_mss\_mem2\_lat

Description: This specifies the maximum latency of memory region 2 .

Units:

Default value: 0

Possible values: 0, 8, 16, 32, 64, 128, 256, 512, 1024

Option: -alb\_mss\_mem2\_def\_lat

Description: This specifies the latency after reset of memory region 2.

Units:

Default value: 0

Minimum value: 0

Maximum value: 1024

Option: -alb\_mss\_mem3\_base\_addr

Description: This specifies the base address of memory region 3 in the memory map, divided by 4KB i.e. do no specify the lower 12 bits of the address.

Units:

Default value: 1048576

Minimum value:

Maximum value:

Option: -alb\_mss\_mem3\_size  
 Description: This specifies the memory size of memory region 3.  
 Units:  
 Default value: 1MB  
 Possible values: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

Option: -alb\_mss\_mem3\_attr  
 Description: This specifies the memory access attribute of memory region 3.  
 Units:  
 Default value: Read-Write  
 Possible values: Read-only, Write-only, Read-Write

Option: -alb\_mss\_mem3\_secure  
 Description: This specifies the memory secure attribute of memory region 3.  
 Units:  
 Default value: Non-Secure  
 Possible values: Non-Secure, Secure

Option: -alb\_mss\_mem3\_lat  
 Description: This specifies the maximum latency of memory region 3.  
 Units:  
 Default value: 0  
 Possible values: 0, 8, 16, 32, 64, 128, 256, 512, 1024

Option: -alb\_mss\_mem3\_def\_lat  
 Description: This specifies the latency after reset of memory region 3.  
 Units:  
 Default value: 0  
 Minimum value: 0  
 Maximum value: 1024

Option: -alb\_mss\_mem\_is\_default\_slave  
 Description: If true then all transactions without destination will be routed here.  
 Units:  
 Default value: true  
 Possible values: true, false

Option: -alb\_mss\_mem\_default\_space  
 Description: This specifies the memory space width of the system.  
 Units:  
 Default value: 32  
 Possible values: 32, 40

---

## D.40 COMPONENT: com.arc.hardware.implementation.1\_0

Option: -clock\_speed  
 Description: Target clock speed of the system  
 Units: MHz

```

Default value: 200
Minimum value: 1
Maximum value: 2000

Option: -ddr2_clk_ratio
Description: DDR2 Clock Vs System Clock Ratio

Units:
Default value: 3x
Possible values: 2x, 3x, 4x

Option: -clock_skew
Description: The clock skew for the system
Units: ns
Default value: 0.2
Minimum value: 0.0
Maximum value: 10.0

Option: -hold_margin
Description: Margin for hold time checks
Units: ns
Default value: 0.05
Minimum value: 0.0
Maximum value: 0.2

Option: -floorplan
Description: Floorplan definition for relative placement of RAMs (at
CPU-level) or the placement of the rams and CPU hard cores (at multicore level)
Units:
Default value: create
Possible values: none, create, User, ARCv2EM4, ARCv2EM4_min,
ARCv2EM4_full, ARCv2EM6, ARCv2EM6_min, ARCv2EM6_full, em4_sensor, em4_rtos, em4, em6,
em6_gp, em4_ecc, em4_parity, em4_em5d_plus, em5d_mini_plusx2, em6_mini_plusx2,
em4_mini_plusx2, em9d_yccm, em9d_xyccm, em11d_yccm

Option: -jtag_tclk
Description: Select the frequency of the JTAG clock Tck (in MHz).

```

The JTAG clock speed has to be less than 1/2 of the cpu clock otherwise the signals on the BVCI interface are not guaranteed to be valid.

**NOTE:** The RTL simulations will work when the JTAG clock frequency is set to half the CPU clock, however this may not be the case when simulating at gate level due to delays on the IO pads.

The default is set to 10 MHz so that there is no conflict when simulating with an ARCanegel3 at 30MHz. (30 > 10\*2)

The speed of simulation can be greatly increased by using a faster JTAG clock, but a dependency will warn if it exceeds 1/2 of the cpu clock.

```

Units: MHz
Default value: 10

```

```
    Minimum value: 1
    Maximum value: 1000

    Option: -execution_trace_level
    Description:
        This traces committed instructions as they execute, and gathers statistics
        visible in the debugger for counting instructions & cycle delays.
        At the "stats" level only the statistics are gathered and no trace is
printed.
        "file" is equivalent to "full", but the results go to a trace .txt file
instead.

    Units:
    Default value: stats
    Possible values: off, stats, full, file

    Option: -generate_ipxact
    Description:
        Generate ipxact.xml file describing the CPUisle or archipelago frontier

    Units:
    Default value: false
    Possible values: true, false

    Option: -ipxact_relative_path_names
    Description:
        Use relative path names for Verilog files in the ipxact. Otherwise, absolute
path names are used.

    Units:
    Default value: true
    Possible values: true, false

    Option: -optional_encryption
    Description:
        When selected, encrypted RTL output is generated.

    Units:
    Default value: false
    Possible values: true, false

    Option: -ignore_encrypt_license
    Description:
        When selected, pretend the encryption license is missing. For testing.

    Units:
    Default value: false
    Possible values: true, false

    Option: -ignore_clear_license
    Description:
        When selected, pretend the cleartext license is missing. For testing.
```

Units:  
Default value: false  
Possible values: true, false



**E**

# Auxiliary Functions for DSP and XY Extensions

## E.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;
```

## E.2 Data Transformation

### E.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>
    realign(bit_vector<d_w> even,           // return realigned data
            bit_vector<d_w> odd,             // data from even memory
            bit_vector<d_w_log2-2> alsb) { // data from odd memory
        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);
}
```

### E.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>
    datamask(bit_vector<d_w> datain,          // masked data output
              bit_vector<2> vw,                // input data
              bit_vector<2> cs,                // vector width, see section 8.3.6
              bit_vector<2> ds) {             // container 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;
}
```

### E.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>
    data2container(bit_vector<d_w> datain,      // expanded data output
                  bit_vector<2> cs,            // input data
                  bit_vector<2> ds) {       // container size, see Table 26
                  // 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;
}

```

## E.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;
}

```

## E.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;
}

```

## E.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];
}

```

## E.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;
}

```

### E.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;
}

```

### E.3.2 Destination Transformation for Endian Parameters

```

// Destination transformation, for endian
// parameters:
// d_w: XY datapath width
bit_vector<2*d_w> dest2endian(bit_vector<2*d_w> datain,           // results can be 64b
                                bit_vector<2>      ds,             // input data, 64b
                                bit                 endian); // data size
// 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;
}

```

# Index

## A

- ABS [370](#)
- absolute instruction [370](#)
- actionpoint
  - access requirements [1057](#)
  - auxiliary registers [1055](#)
  - control register [1062](#)
  - extensions to debug register [1048, 1059](#)
  - host and core register access [1058](#)
  - match mask register [1061](#)
  - registers reset state [1058](#)
  - system [1055](#)
- actionpoint configuration register [1060](#)
- active interrupts register [134](#)
- ADC [372](#)
- ADD [1254](#)
  - add instruction
    - syntax summary [269](#)
  - add with carry instruction [372](#)
  - add with scaled source instruction
    - shift 1 [376](#)
    - shift 2 [378](#)
    - shift 3 [380](#)
- ADD1 [376](#)
- ADD2 [378](#)
- ADD3 [380](#)
- address space [79](#)
- addressing modes [92](#)
  - compare and branch [94](#)
  - conditional branch [93](#)
  - conditional execution [93](#)
  - null instruction format [93](#)
- AE flag [121](#)
- AEX [382](#)
- AND [384](#)
- and instruction [384](#)
- AP\_BUILD [1060](#)
- ARCNUM [112](#)
- ARCv2
  - introduction [63](#)
  - key features [63](#)
- arithmetic shift left instruction [386](#)
- arithmetic shift right instruction [391](#)
  - multiple [393, 396](#)
- ASL [386](#)
- ASL multiple [388](#)
- ASR [391](#)
- ASR multiple [393, 396](#)
- atomic exchange instruction [517](#)
  - syntax summary [280](#)
- AUX\_CACHE\_LIMIT [982](#)
- AUX\_DCCM [960](#)
- AUX\_ICCM [940](#)
- AUX\_IRQ\_ACT [134, 891, 893](#)
- AUX\_IRQ\_CTRL [126](#)
- AUX\_IRQ\_HINT [136](#)
- AUX\_RTC\_CTRL [1024](#)
- AUX\_RTC\_HIGH [1027](#)
- AUX\_RTC\_LOW [1026](#)
- AUX\_USER\_SP [125, 130, 131, 132, 133](#)
- auxiliary register instructions
  - summary [283](#)
- auxiliary register map [928, 993, 1003, 1134, 1141, 1165](#)
- auxiliary register set [928, 959, 993, 1003, 1134, 1141, 1165](#)
  - 64-bit rtc [1024](#)
- actionpoint configuration [1060](#)
- active interrupts [134](#)
- barrel shifter configuration [178](#)
- BCR version [168, 182](#)
- branch target address [162](#)

BTA configuration 169  
build configuration 166, 983, 1007, 1114, 1127, 1161, 1236, 1712  
core register set configuration 172  
DCCM base address 960  
DCCM RAM configuration 961  
exception cause 147  
exception fault address 159  
exception return address 144  
exception return branch target address 145, 1074  
exception return status 146  
execute indexed base address 143  
ICCM configuration 941  
identity 111  
individual interrupt pending 165  
instruction cache build 957  
instruction fetch queue configuration 943  
interrupt build configuration 182  
interrupt cause 154  
interrupt context saving control 126  
interrupt enable 156  
interrupt P0 status link 124  
interrupt priority 137  
interrupt priority pending 135, 894  
interrupt pulse cancel 164  
interrupt select 155  
interrupt status 158  
interrupt trigger 157  
interrupt vector base 128, 129, 130, 131, 132, 133, 134  
interrupt vector base address configuration 170  
ISA configuration 179  
JLI\_BASE 141, 166  
LDI\_BASE 142  
loop end 110  
loop start 109  
memory protection unit build configuration 921  
memory protection unit ECR 916  
memory protection unit enable 913  
MIN/MAX configuration 177  
MPU region descriptor base 918  
MPU region descriptor permissions 919  
multiply configuration 174  
non cached memory region 982  
normalize configuration 176  
peripheral memory region 1046  
processor timers 1009  
processor timers configuration 1028  
program counter 113  
rtc count high 1027  
rtc count low 1026  
saved user stack pointer 125, 130  
smart control 1077  
software interrupt 136  
stack region build configuration 934  
status 117  
summary 66  
swap configuration 175  
timer 0 control 1013, 1019  
timer 0 count 1012, 1018  
timer 0 limit 1014, 1020  
timer 1 control 1016, 1022  
timer 1 count 1015, 1021  
timer 1 limit 1017, 1023  
user mode extension enable 159  
auxiliary registers 283

**B**

B 403  
B\_S 405  
barrel shift instructions  
    syntax summary 272  
barrel shifter configuration register 178  
BARREL\_BUILD 178  
basecase ALU instructions  
    summary 268, 269, 270  
BBIT0 407  
BBIT1 409  
Bcc 411  
Bcc\_S 413  
BCLR 415  
BCR version register 168, 182, 934  
BCR\_VER 168, 181, 1107  
BH bit 1050  
BI 417  
BIC 419  
BIH 421  
bit clear instruction 415  
bit exclusive-or instruction 453  
bit mask instruction 426, 429  
bit set instruction 439  
bit test instruction 450  
bit toggle instruction 453  
bitwise and with invert instruction 419  
BLcc 423  
BLINK 100

- BMSK [426](#), [429](#)  
 branch and link instruction [423](#)  
 branch and link instructions  
   syntax summary [286](#)  
 branch conditionally instruction [411](#)  
   16-bit format [413](#)  
 branch indexed [417](#)  
 branch indexed to half-word [421](#)  
 branch instructions  
   summary [285](#)  
   syntax summary [285](#)  
 branch on bit test instruction  
   clear [407](#)  
   syntax summary [287](#)  
 branch on bit test set instruction  
   set [409](#)  
 branch on compare instruction [432](#)  
 branch on compare instructions  
   syntax summary [287](#)  
 branch target address register [162](#)  
 branch unconditionally instruction [403](#)  
 BRcc [432](#)  
 breakpoint instruction [436](#), [852](#)  
   summary [290](#)  
   trap 0 [803](#)  
 BRK [436](#)  
 BSET [439](#)  
 BTA [162](#)  
 BTA configuration register [169](#)  
 BTA\_LINK\_BUILD [169](#)  
 BTST [450](#)  
 build configuration register  
   actionpoint [1060](#)  
   barrel shifter [178](#)  
   BCR version [168](#)  
   BTA [169](#)  
   core register set [172](#)  
   DCCM RAM [961](#)  
   ICCM [941](#)  
   instruction cache build [957](#)  
   instruction fetch queue [943](#)  
   interrupt [182](#)  
   interrupt vector base address [170](#)  
   ISA [179](#)  
   memory protection unit [921](#)  
   MIN/MAX [177](#)  
   multiply [174](#)  
   normalize [176](#)  
   processor timers [1028](#)  
   stack region [934](#)  
   swap [175](#)  
 build configuration registers [166](#), [983](#), [1007](#), [1114](#),  
[1127](#), [1161](#), [1236](#), [1712](#)  
 BXOR [453](#)
- C**
- canceling pulse triggered interrupts [197](#)  
 CLRI [455](#)  
 CMP [457](#)  
 compare and branch instruction [432](#)  
 compare instruction [457](#)  
   reverse compare [729](#)  
 condition codes [267](#)  
 conditional branch [93](#), [94](#)  
 conditional branch and link instruction [423](#)  
 conditional branch instruction [411](#)  
   16-bit format [413](#)  
 conditional execution [93](#)  
 conditional jump and link instruction [614](#)  
 conditional jump instruction [608](#)  
 CONTROL0 [1013](#), [1019](#)  
 core register map [95](#), [1117](#)  
 core register set [95](#), [1117](#)  
   extension registers [102](#), [1118](#), [1227](#)  
   immediate data indicator [101](#)  
   link registers [100](#)  
   loop count [100](#)  
   pointer registers [99](#)  
   reduced configurations [98](#)  
   reserved register r61 [101](#)  
   summary [66](#)  
   used in 16-bit instructions [97](#)  
 core register set configuration register [172](#)  
 COUNT0 [1012](#), [1018](#)
- D**
- data formats [80](#)  
   16-bit data [89](#)  
   1-bit data [90](#)  
   32-bit data [88](#)  
   8-bit data [90](#)  
 DCCM\_BUILD [961](#)  
 DCCM\_BUILD configuration register [961](#)

DEBUG 1048  
debug register 1048  
delayed load from memory instruction 624  
DIV 473, 1316  
DIVU 363, 365, 476, 479, 482, 488, 687, 815, 818, 827, 830, 833, 836, 844, 1240, 1241, 1322, 1325, 1334, 1340, 1352, 1355, 1358, 1361, 1434, 1517, 1557, 1566, 1569, 1623, 1634, 1684, 1687, 1690, 1696, 1699  
DMP\_PERIPHERAL 1046  
double fault 244  
DZ bit 122  
**E**  
ECR 147  
EI\_BASE 143  
EI\_S 121, 491, 511  
emulation of extension condition codes 249  
emulation of extension instructions 249  
emulation of extension registers 249  
encoding immediate data 101  
ENTER\_S 513  
ERBTA 145  
ERET 144  
ERSTATUS 146  
EX 517  
exception cause register 147  
exception fault address register 159  
exception handling 245  
exception return address register 144  
exception return branch target address register 145, 1074  
exception return status register 146  
exceptions  
    cause codes 223  
    cause register 221, 223  
    data TLB miss 241  
    data TLB protection violation 241  
    delay slots 247  
    double fault 244  
    emulation of extension condition codes 249  
    emulation of extension instructions 249  
    emulation of extension registers 249  
    entry 244  
    exit 246  
    extension instruction 241

illegal instruction sequence 238  
instruction error 237  
instruction fetch protection violation 235, 242  
instruction fetch TLB miss 241  
machine check, double fault 234  
machine check, fatal TLB error 235  
machine check, instruction fetch memory error 241  
machine check, overlapping TLB entries 235  
overview 220  
parameters 223  
precision 220  
priorities 224  
privilege violation, action point hit on instruction fetch 237  
privilege violation, actionpoint hit memory or register 243  
privilege violation, disabled extension 241  
privilege violation, kernel only access 241  
privilege violation, misaligned data access 241  
reset 234  
types 224  
vector name 221  
vector number 221  
vector offset 221  
vectors 221  
    with interrupts 244  
exchange instruction 517  
    syntax summary 280  
exclusive-or instruction 846  
EXTB 520  
extend instruction  
    signed byte 770  
    signed half-word 772  
    unsigned byte 520  
    unsigned half-word 522  
extension core registers 102, 1118, 1227  
extension instructions  
    summary 292  
extensions  
    auxiliary registers 76  
    condition codes 76  
    core register 76  
    instruction set 76  
    summary 76  
extensions instruction  
    syntax summary 292  
EXTH 522

**F**

features 63  
 FFS 566  
 FH bit 850, 1048  
 find first set 566  
 find last set 573  
 FLAG 568, 774  
 flag instruction 568, 774  
 FLS 573  
 force halt 1048  
 FP 99  
 function epilogue sequence 631  
 function prologue sequence 513

**G**

GP 99

**H**

H bit 123  
 halting the processor 850  
 host interface 849  
**I**  
 I\_CACHE\_BUILD 957  
 I\_CACHE\_BUILD configuration register 957  
 ICAUSE 154  
 ICCM\_BUILD 941  
 ICCM\_BUILD configuration register 941  
 IDENTITY 111  
 identity register 111  
 IE bit 119  
 IFQUEUE\_BUILD 943  
 IFQUEUE\_BUILD configuration register 943  
 ILINK 100  
 immediate data indicator 101  
 instruction error 237  
 instruction format

summary 90

instruction set  
 16 and 32 bit summary 66  
 instruction syntax convention 251  
 INT\_VECTOR\_BASE 128  
 interrupt  
 pending interrupts 194  
 interrupt (P0) status link registers 124  
 interrupt cause registers 154, 197

interrupt context saving control 126  
 interrupt priority pending register 135  
 interrupt unit configuration 191  
 interrupt unit programming 193  
 interrupt vector base address configuration register 170  
 interrupt vector base register 128, 129, 130, 131, 132, 133, 134  
 interrupts  
 canceling pulse triggered interrupts 197  
 cause registers 197  
 configuration 191  
 programming 193  
 returning 211  
 sensitivity configuration 196  
 software triggered 197  
 timing 219  
 vector base address configuration 192

IRQ\_BUILD 182  
 IRQ\_BUILD configuration register 182  
 IRQ\_ENABLE 156  
 IRQ\_PENDING 165  
 IRQ\_PRIORITY 137  
 IRQ\_PRIORITY\_PENDING 135, 894  
 IRQ\_PULSE\_CANCEL 164  
 IRQ\_SELECT 155  
 IRQ\_STATUS 158  
 IRQ\_TRIGGER 157  
 IS bit 851  
 ISA configuration register 179  
 ISA\_CONFIG 179

**J**

J 605  
 Jcc 608  
 JL 611  
 JLcc 614  
 JLI\_BASE 139, 140, 141, 166, 1135, 1136  
 JLI\_S 617  
 jump and link conditionally instruction 614  
 jump and link indexed 617  
 jump and link unconditionally instruction 611  
 jump conditionally instruction 608  
 jump instructions  
 summary 288

jump unconditionally instruction 605

## K

kernel flag instruction 619

kernel mode 184

KFLAG 619

## L

L flag 122

LD 624

LD bit 1050

LDI 629

LDI\_BASE 142

LEAVE\_S 631

limm 101

link register 199

link registers 100

list of Instructions 361, 1239

LLOCK 635, 637

load from auxiliary register instruction 646

    syntax summary 283

load from memory instruction 624

load pending 850, 1050

load register 283

load/store instructions

    summary 277

logical operations 292

logical shift left 650

logical shift left 16 648

logical shift right 659

logical shift right 16 657

logical shift right instruction 652

    multiple 654

Long immediate data

    ALU operations 292

    indicator 97

long immediate data

    16-bit instructions 344

    32-bit instructions 259, 362, 373, 377, 379, 381,

        385, 390, 395, 397, 416, 420, 428, 430,

        440, 449, 452, 454, 475, 478, 480, 484,

        485, 487, 490, 491, 526, 529, 532, 535,

        540, 543, 546, 549, 552, 555, 558, 565,

        579, 582, 585, 588, 591, 594, 597, 604,

        628, 656, 663, 666, 669, 672, 675, 678,

        683, 686, 689, 692, 695, 697, 699, 701,

714, 727, 728, 729, 733, 736, 747, 753, 759, 765, 789, 792, 794, 796, 798, 812, 814, 817, 820, 823, 825, 829, 832, 835, 838, 847, 1248, 1251, 1253, 1256, 1263, 1271, 1274, 1278, 1282, 1286, 1289, 1293, 1296, 1300, 1303, 1308, 1312, 1315, 1318, 1321, 1324, 1327, 1330, 1333, 1336, 1339, 1342, 1343, 1345, 1348, 1351, 1354, 1357, 1360, 1363, 1364, 1374, 1377, 1380, 1383, 1386, 1389, 1392, 1395, 1398, 1401, 1404, 1407, 1410, 1413, 1416, 1425, 1430, 1433, 1436, 1439, 1442, 1445, 1448, 1451, 1454, 1459, 1464, 1467, 1470, 1473, 1476, 1479, 1482, 1485, 1488, 1491, 1494, 1497, 1500, 1503, 1506, 1509, 1512, 1516, 1519, 1520, 1521, 1524, 1527, 1530, 1533, 1535, 1538, 1543, 1545, 1547, 1550, 1552, 1556, 1559, 1562, 1565, 1568, 1571, 1573, 1576, 1579, 1582, 1585, 1588, 1590, 1592, 1594, 1596, 1599, 1602, 1606, 1610, 1613, 1616, 1619, 1622, 1625, 1629, 1633, 1636, 1640, 1644, 1648, 1651, 1653, 1656, 1660, 1662, 1664, 1666, 1668, 1670, 1672, 1675, 1677, 1679, 1681, 1683, 1686, 1689, 1692, 1695, 1698, 1701

loop count register 100

loop instruction

    syntax summary 288

loop set up instruction 471, 639

loops, zero-overhead 249

LP instruction 288

LP\_COUNT 100, 288

LP\_END 110, 288

LP\_START 109, 288

LPcc 471, 639

LR 646

LSL16 648

LSL8 650

LSR 652

LSR multiple 654

LSR16 657

LSR8 659

## M

MAX 673

maximum instruction 673

memory based instructions

summary 277  
 memory error  
     instruction fetch 241  
 memory prefetch instruction 720  
 MIN 676  
 MIN/MAX configuration register 177  
 minimum instruction 676  
 MINMAX\_BUILD 177  
 MOV 679  
     move instruction 679, 1417, 1421  
 MPU\_BUILD 921  
 MPU\_BUILD configuration register 921  
 MPU\_ECR 916  
 MPU\_EN 885, 913, 1038  
 MPY 681, 1423  
 MPYH 690, 1443  
 MPYHU 693, 1446  
 MPYU 696, 1449  
 MPYUW 698, 1452  
 MPYW 700, 1457  
 multiple arithmetic shift right instruction 393, 396  
 multiply  
     high result, signed 1443  
     high result, unsigned 693, 1446  
     signed 16x16 with 32-bit result 700, 1457  
     unsigned 16x16 with 32-bit result 698, 1452  
 multiply configuration register 174  
 MULTIPLY\_BUILD 174

**N**

NEG 702  
 negate instruction 702  
 no operation instruction 704  
 NOP\_S 704  
 NORM 706  
 NORM\_BUILD 176  
 normalize configuration register 176  
 normalize instruction 706  
     word 708  
 NORMW 708  
 NOT 711  
 not instruction 711  
 null instruction format 93

**O**

operating modes 183  
     summary 67  
     switching 187  
 operating systems 245  
 OR 713  
 or instruction 713

**P**

PC 113  
 PCL 101  
 plotting results 1099  
 pointer registers 99  
 pop from stack instruction 716  
 POP\_S 716  
 precise exceptions 220  
 PREFETCH 720  
 prefetch from memory instruction 720  
     syntax summary 281  
 privilege violations 184  
 privileged instructions 185  
 privileged registers 187  
 privileges  
     introduction 183  
     overview 184  
 processor timers configuration register 1028  
 program counter register 113  
 programmer's model 66, 1216  
 push onto stack instruction 726  
 PUSH\_S 726

**R**

RA bit 1050  
 RCMP 729  
 register bank 1052  
 REM 731, 1521  
 REMU 734, 1525  
 reserved register r61 101  
 reset 234  
 return from interrupt/exception instruction 754  
     summary 289  
 returning from interrupts 211  
 reverse compare instruction 729  
 reverse subtract instruction 752  
 RF\_BUILD 172

RLC 737  
ROL 739  
ROL8 741  
ROR 743  
ROR multiple 745  
ROR8 748  
rotate left 739  
rotate left through carry instruction 737  
rotate left8 741  
rotate right 8 748  
rotate right instruction 743  
    multiple 745  
rotate right through carry instruction 750  
RRC 750  
RSUB 752  
RTIE 754  
RTOS 245

**S**

SBC 758  
SCOND 760, 762  
self halt 1050  
SETcc 764  
SETEQ 766  
SETI 767  
SEXB 770  
SEXH 772  
SH bit 1050  
sign extend instruction  
    byte 770  
    half-word 772  
signed 16x16 multiply  
    with 32-bit result 700, 1457  
signed multiply  
    high result 690, 1443  
single instruction step 851, 1049  
single operand extension instruction  
    syntax summary 293  
single step 851  
SLEEP 776, 839  
sleep instruction 776, 839, 1050  
    summary 290  
SMART\_CONTROL 1077  
software interrupt instruction 803  
software interrupt register 136

software triggered interrupt 197  
SP 99  
SR 784  
ST 786  
STACK\_REGION\_BUILD 934  
STACK\_REGION\_BUILD configuration register 934  
starting the processor 851  
status register 117  
status save register 199  
STATUS32 117  
store register 283  
store to auxiliary register instruction 784  
    syntax summary 283  
store to memory instruction 786  
    syntax summary 279  
SUB 790  
SUB1 793  
SUB2 795  
SUB3 797  
subtract instruction 790  
    reverse 752  
subtract with carry instruction 758  
subtract with scaled source instruction  
    shift 1 793  
    shift 2 795  
    shift 3 797  
SWAP 399, 401, 799  
swap byte ordering 801  
swap configuration register 175  
swap instruction 399, 401, 799  
SWAP\_BUILD 175  
SWAPE 801  
SWI 803  
switching operating modes 187  
SYNC 806  
synchronize instruction 806  
    summary 291  
syntax summary  
    add instruction 269  
    atomic exchange instruction 280  
    auxiliary register instructions 283  
    barrel shift instructions 272  
    branch and link instructions 286  
    branch instructions 285  
    branch on bit test instruction 287

branch on compare instructions 287  
exchange instruction 280  
extension instructions 292  
jump instructions 288  
load from auxiliary register instruction 283  
load/store instructions 277  
loop instruction 288  
memory based instructions 277  
prefetch from memory instruction 281  
single operand extension instruction 293  
store to auxiliary register instruction 283  
store to memory instruction 279  
zero operand extension instruction 293  
zero operand instructions 289

**T**

test instruction 811  
timer 0 control register 1013, 1019  
timer 0 count register 1012, 1018, 1026, 1027  
timer 0 limit register 1014, 1020  
timer 1 control register 1016, 1022  
timer 1 count register 1015, 1021  
timer 1 limit register 1017, 1023  
Timer COUNT0 1012  
TIMER\_BUILD 1028  
trap instruction 809  
    summary 291  
    trap 0 803  
TRAP\_S 809  
TRAP0 803  
TST 811  
typographic convention 63

**U**

U flag 122  
UB bit 1050  
unconditional branch instruction  
    16-bit format 405  
unconditional jump and link instruction 611  
UNIMP\_S 813  
unimplemented instruction 813  
unsigned 16x16 multiply  
    with 32-bit result 698, 1452  
unsigned multiply  
    high result 693, 1446  
user mode 184

user mode extension enable register [159](#)

## V

VECBASE\_AC\_BUILD [170](#), [171](#)

## X

XOR [846](#)

## Z

zero delay loops [100](#), [288](#)

zero extend instruction

    byte [520](#)

    half-word [522](#)

zero operand extension instruction

    syntax summary [293](#)

zero operand instructions

    summary [289](#)

    syntax summary [289](#)

zero-overhead loop [249](#)

ZZ bit [1050](#)