



DesignWare® ARCV2 ISA

**Programmer's Reference Public Edition Manual for ARC HS
Processors**

Copyright Notice and Proprietary Information

© 2023 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

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

Trademarks

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

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

Synopsys, Inc.
www.synopsys.com

Contents

List of Figures	31
List of Examples	51
List of Tables	53
Preface	69
Document Organization	69
Document Revision History	70
Differences Between ARC HS3x and ARC HS4x Processors	70
Synopsys Statement on Inclusivity and Diversity	71
Quick Reference	73
Chapter 1	
Build and Auxiliary Register List	75
1.1 Build Configuration Registers	84
Chapter 2	
Exceptions List	87
Part 1	
ARCV2 Baseline ISA	93
Chapter 2	
Introduction	95
2.1 Typographic Conventions	95
2.2 Key Features of the ARCV2 ISA	95
2.3 Programmer's Model	97
2.3.1 Core Register Set	97
2.3.2 Auxiliary Register Set	97
2.3.3 32-bit Instruction Formats	97
2.3.4 16-bit Instruction Formats	98
2.3.5 Operating Privileges	98
2.4 Configurability	99
2.4.1 Programming Model Options	99
2.4.2 Processor Component Options	101
2.4.3 ISA Options	102
2.5 Custom Extensions	106
2.5.1 Extension Core Registers	106

2.5.2	Extension Auxiliary Registers	106
2.5.3	Extension Instructions	106
2.5.4	Extension Condition Codes	106
 Chapter 3		
Data Organization and Addressing		109
3.1	The ARCV2 Address Space	109
3.2	Data Formats	110
3.2.1	Vector Operands	110
3.2.2	32-Bit Operand Data Elements	112
3.2.3	Expansion of Literals	112
3.2.4	Expansion of Literals in Vector Instructions	112
3.2.5	Expansion of Literals in Store Double (STD)	113
3.3	Floating-Point Data Formats	113
3.3.1	Half-Precision Floating-Point Data Formats	114
3.3.2	Single-Precision Floating-Point Data Formats	114
3.3.3	Double-Precision Floating-Point Data Formats	115
3.3.4	NaN Formats	115
3.4	Data Layout in Memory	116
3.4.1	64-Bit Data	117
3.4.2	32-Bit Data	118
3.4.3	16-Bit Data	118
3.4.4	8-Bit Data	119
3.4.5	1-Bit Data	119
3.5	Instruction Layout in Memory	120
3.6	Addressing Modes	121
3.7	Null Instruction Format	122
3.8	Conditional Execution	122
3.9	Conditional Branch Instruction	123
3.10	Compare and Branch Instruction	123
 Chapter 4		
Core Architectural State Registers		125
4.1	Core Register Set	125
4.1.1	Core Register Mapping Used in 16-bit Instructions	127
4.1.2	Reduced Configuration of Core Registers	128
4.1.3	Multiple Register Banks	129
4.1.4	Illegal Core Register Usage	131
4.1.5	Pointer Registers, GP (r26), FP (r27), SP (r28)	131
4.1.6	Link Registers, ILINK (r29), BLINK (r31)	131
4.1.7	Long Immediate Data Indicator Register, (r30)	132
4.1.8	Loop Count Register, LP_COUNT (r60)	132
4.1.9	Reserved Register (r61)	132
4.1.10	Immediate Data Indicator, L IMM (r62)	132
4.1.11	Word-aligned Program Counter, PCL (r63)	133
4.2	Error Protection on Core Registers	133
4.3	Extension Core Registers	133
4.3.1	Illegal Extension Core Register Usage	134
4.4	Auxiliary Register Set	134

4.4.1	Baseline Auxiliary Registers	134
4.4.2	Register Access Permissions	135
4.4.3	Optional Instruction Set Auxiliary Registers	136
4.4.4	User Extension Auxiliary Registers	137
4.4.5	Optional Build Configuration Registers	138
4.4.6	Reset Value of Auxiliary Registers	138
4.4.7	Auxiliary Registers Holding Address Values	138
4.4.8	Loop Start Register, LP_START	139
4.4.9	Loop End Register, LP_END	140
4.4.10	Core Identity Register, IDENTITY	141
4.4.11	Program Counter, PC	143
4.4.12	Status Register, STATUS32	145
4.4.13	Status Register Priority 0, STATUS32_P0	151
4.4.14	Saved User Stack Pointer, AUX_USER_SP	152
4.4.15	Interrupt Context Saving Control Register, AUX_IRQ_CTRL	153
4.4.16	Interrupt Vector Base Register, INT_VECTOR_BASE	154
4.4.17	Active Interrupts Register, AUX_IRQ_ACT	155
4.4.18	Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING	156
4.4.19	Software Interrupt Trigger, AUX_IRQ_HINT	157
4.4.20	Interrupt Priority Register, IRQ_PRIORITY	158
4.4.21	Data Memory Attribute Register, DATA_MEM_ATTR	160
4.4.22	Jump and Link Indexed Base Address, JLI_BASE	162
4.4.23	Load Indexed Base Address, LDI_BASE	163
4.4.24	Execute Indexed Base Address, EI_BASE	164
4.4.25	Exception Return Address, ERET	165
4.4.26	Exception Return Branch Target Address, ERBTA	166
4.4.27	Exception Return Status, ERSTATUS	167
4.4.28	Exception Cause Register, ECR	168
4.4.29	Exception Fault Address, EFA	170
4.4.30	Interrupt Cause Registers, ICAUSE	172
4.4.31	Interrupt Select, IRQ_SELECT	173
4.4.32	Interrupt Enable Register, IRQ_ENABLE	174
4.4.33	Interrupt Trigger Register, IRQ_TRIGGER	175
4.4.34	Interrupt Status Register, IRQ_STATUS	176
4.4.35	User Mode Extension Enable Register, XPU	177
4.4.36	Branch Target Address, BTA	179
4.4.37	Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL	181
4.4.38	Interrupt Pending Register, IRQ_PENDING	182
4.4.39	User Extension Flags Register, XFLAGS	183
4.5	Build Configuration Registers	183
4.5.1	Build Configuration Registers Version, BCR_VER	185
4.5.2	BTA Configuration Register, BTA_LINK_BUILD	186
4.5.3	Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD	187
4.5.4	Core Register File Configuration Register, RF_BUILD	189
4.5.5	Multiplier Configuration Register, MULTIPLY_BUILD	191
4.5.6	Swap Instruction Configuration Register, SWAP_BUILD	192
4.5.7	Normalize Instruction Configuration Register, NORM_BUILD	193
4.5.8	Min/Max Instruction Configuration Register, MINMAX_BUILD	194
4.5.9	Barrel Shifter Configuration Register, BARREL_BUILD	195

4.5.10 Instruction Set Configuration Register, ISA_CONFIG	196
4.5.11 Interrupt Build Configuration Register, IRQ_BUILD	199
4.5.12 ARCV2 ISA Profile Register, ISA_PROFILE	201
4.5.13 Micro Architecture Configuration Register, MICRO_ARCH_BUILD	203
Chapter 5	
Interrupts and Exceptions	205
5.1 Introduction	205
5.2 Privileges and Operating Modes	205
5.2.1 Kernel Mode	205
5.2.2 User Mode	205
5.2.3 Privilege Violations	206
5.2.4 Switching Between Operating Modes and Privilege Levels	207
5.3 Interrupts	208
5.3.1 Interrupt Unit Features	208
5.3.2 Interrupt Unit Configuration	209
5.3.3 Interrupt Unit Programming	211
5.3.4 Interrupt Handling	216
5.4 Exceptions	226
5.4.1 Exception Precision	226
5.4.2 Exception Vectors and the Exception Cause Register	228
5.4.3 Exception Types and Priorities	230
5.4.4 Exception Detection	249
5.4.5 Effect of Exceptions and Interrupts on Operating Mode	249
5.4.6 Exception Entry	249
5.4.7 Exception Exit	251
5.4.8 Exceptions and Delay Slots	252
5.4.9 Emulation of Extension Instructions	254
5.4.10 Emulation of Extension Registers and Condition Codes	254
Chapter 6	
Instruction Set Summary	255
6.1 Instruction Set Encoding	255
6.1.1 Top-level Instruction Formats	255
6.1.2 Instruction Set Profiles	257
6.1.3 Assignment of Instruction Formats to Major Opcodes	258
6.1.4 32-bit Instruction Formats	260
6.1.5 16-bit Instruction Formats	261
6.1.6 Encoding Notation	262
6.1.7 Long Immediate Source Operands and Null Destination Operands	263
6.1.8 Condition Code Tests	264
6.1.9 Encoding Branch and Jump Delay Slot Modes	266
6.1.10 Encoding Static Branch Predictions	266
6.1.11 Encoding Load / Store Address Writeback Modes	267
6.1.12 Encoding Load / Store Cache Bypass Mode	268
6.1.13 Encoding Load / Store Data Sizes	268
6.1.14 Encoding Load Data Extension Modes	268
6.1.15 Encoding Store Data Source Mode	269
6.1.16 Use of Reserved Encodings	269

6.1.17 Use of Illegal Encodings	269
6.2 Instruction Syntax Conventions	270
6.3 Status flags and conditions	271
6.3.1 Flag Setting	272
6.3.2 Status Flags Notation	272
6.4 Arithmetic and Logical Instructions	272
6.4.1 Integer Adder Operations	273
6.4.2 Move Operations	274
6.4.3 Bit-wise Logical Operations	275
6.4.4 Bit Operations and Mask Operations	275
6.4.5 Shift and Rotate Operations	276
6.4.6 Selection Operations	277
6.4.7 Byte-swapping Operations	278
6.4.8 Bit-scanning Operations	278
6.4.9 Relational Comparison Operations	278
6.4.10 Integer Multiply, Multiply-accumulate, and Divide Operations	279
6.4.11 Dual and Quad Integer Multiply / Accumulate Operations	280
6.4.12 Integer Vector Operations	281
6.5 Memory Access Instructions	283
6.5.1 Load Instructions	284
6.5.2 Store Instructions	285
6.5.3 Stack Pointer Operations	285
6.5.4 Atomic Memory Operations	286
6.5.5 Prefetch Instructions	287
6.6 Auxiliary Register Operations	288
6.6.1 Load from Auxiliary Register	289
6.6.2 Store to Auxiliary Register	289
6.6.3 Auxiliary Register Exchange	289
6.7 Control Flow Instructions	289
6.7.1 Branch Instructions	291
6.7.2 Jump Instructions	294
6.7.3 Zero Overhead Loop Instruction	294
6.7.4 Return from Interrupt or Exception Instruction	294
6.8 Special Instructions	294
6.8.1 Breakpoint Instruction	296
6.8.2 Sleep Instruction	296
6.8.3 Software Interrupt Instruction	296
6.8.4 SETI	296
6.8.5 CLRI	296
6.8.6 Trap Instruction	296
6.8.7 Synchronization Instructions	297
6.9 APEX Extension Instructions	297
6.9.1 Syntax for Generic Extension Instructions	297
6.9.2 Syntax for Single Operand Extension Instructions	298
6.9.3 Syntax for Zero Operand Extension Instructions	298
6.10 Floating-Point Instructions	299
6.11 DSP Instructions	302
6.11.1 Basic Saturating Arithmetic Operations	303
6.11.2 Vector Unpacking Operations	304

6.11.3	Vector ALU Operations	305
6.11.4	Accumulator Operations	307
6.11.5	Vector SIMD 16x16 MAC Operations	307
6.11.6	Dual Inner Product 16x16 MAC Operations	309
6.11.7	32x32 and 16x16 MAC Operations	309
6.11.8	Dual 16x8 MAC Operations	311
6.11.9	32x16 MAC Operations	311
6.11.10	Complex Operations	313
6.11.11	Vector 64-Bit Operations	315
6.11.12	Divide and Square-Root Instructions	316
6.11.13	Address Generation Instructions	317

Chapter 7

32-bit Instruction Formats Reference	319
7.1 Branch Format [F32_BR0]	320
7.1.1 Branch-Conditional Sub-format [F32_BR0, COND]	320
7.1.2 Branch Unconditional Far Sub-format [F32_BR0, UCOND_FAR]	320
7.2 BRcc, BBITn and BL Format [F32_BR1]	321
7.2.1 BRcc / BBITn Register-register Format [F32_BR1, BCC, REG_REG]	321
7.2.2 Register-immediate Format [F32_BR1, BCC, REG_U6]	322
7.2.3 Branch-and-link Format [F32_BR1, BL]	323
7.3 Load Register with Offset Format [F32_LD_OFFSET]	325
7.4 Store Register with Offset Format [F32_ST_OFFSET]	326
7.5 General Operations Format [F32_GEN4]	327
7.5.1 Operator Format Indicators	327
7.5.2 Operand Format Indicators	327
7.5.3 Syntax and Encoding of Instructions in General Operations Formats	328
7.5.4 Dual-operand Instructions, F32_GEN4	331
7.5.5 Move and Compare Instructions, 0x04, [0x0A – 0x0D] and 0x04, [0x11]	333
7.5.6 Jump and Jump-and-Link Conditionally, 0x04, [0x20 – 0x23]	333
7.5.7 Load Register-Register, 0x04, [0x30 – 0x37]	333
7.5.8 Single Operand Instructions, F32_GEN4	334
7.5.9 Zero Operand Instructions, F32_GEN4	334
7.6 F32_EXT5	335
7.6.1 Dual-operand Instructions, F32_EXT5	335
7.6.2 Single-operand Instructions, F32_EXT5	336
7.6.3 Zero-operand Instructions, F32_EXT5	337
7.7 F32_EXT6	338
7.7.1 Dual-operand Instructions, F32_EXT6	338
7.7.2 Single-operand Instructions, F32_EXT6	338
7.7.3 Zero-operand Instructions, F32_EXT6	339
7.8 APEX Extension Instruction Format [F32_APEX]	340
7.8.1 Extension ALU Operation, Register with Unsigned 6-bit Immediate	340
7.8.2 Extension ALU Operation, Register with Signed 12-bit Immediate	340
7.8.3 Extension ALU Operation, Conditional Register	340
7.8.4 Extension ALU Operation, Conditional Reg with Unsigned 6-bit Immediate	341
7.8.5 FastMath Extension Pack Instructions	341

Chapter 8

16-bit Instruction Formats Reference	343
8.1 Compact Move/Load [F16_COMPACT]	345
8.2 Compact Load/Add/Sub [F16_LD_ADD_SUB]	346
8.3 Compact Load/Store [F16_LD_ST_1]	347
8.4 Indexed Jump or Execute [F16_JLI_EI]	348
8.5 Load / Add Register-Register [F16_LD_ADD_RR]	349
8.6 Add/Sub/Shift Register-Immediate [F16_ADD_IMM]	350
8.7 Dual Register Operations [F16_OP_HREG]	351
8.7.1 Long Immediate Operands in Format 0x0E	352
8.8 General Register Format Instructions [F16_GEN_OP]	353
8.8.1 DOP-format General Operations	353
8.8.2 SOP-format 16-bit General Operations	354
8.8.3 ZOP-format 16-bit General Operations	355
8.9 16-bit Load and Store Formats with Offset	357
8.10 Shift/Subtract/Bit Immediate [F16_SH_SUB_BIT]	358
8.11 Stack-based Operations [F16_SP_OPS]	359
8.12 Load/Add GP-Relative [F16_GP_LD_ADD]	360
8.13 Load PCL-Relative [F16_PCL_LD]	361
8.14 Move Immediate [F16_MV_IMM]	362
8.15 ADD/CMP Immediate [F16_OP_IMM]	363
8.16 Branch on Compare Register with Zero [F16_BCC_REG]	364
8.17 Branch Conditionally [F16_BCC]	365
8.17.1 Branch Conditionally with cc Field, 0x1E, [0x03, 0x00 – 0x07]	365
8.18 Branch and Link Unconditionally [F16_BL]	367
Chapter 9	
Instruction Set Details	369
9.1 Instruction List	370
9.2 ARC Instructions	379
ABS	380
ADC	382
ADD	384
ADD1	386
ADD2	388
ADD3	390
AEX	392
AND	394
ASL	396
ASL Multiple	398
ASR	401
ASR multiple	403
ASR16	406
ASR8	408
B	410
B_S	412
BBIT0	414
BBIT1	416
Bcc	418
Bcc_S	420

BCLR	422
BI	424
BIC	426
BIH	428
BLcc	430
BMSK	433
BMSKN	436
BRcc	438
BRK	441
BSET	444
BTST	446
BXOR	449
CLRI	451
CMP	453
DBNZ	455
DIV	457
DIVU	460
DMACH	463
DMACHU	465
DMACWH	468
DMACWHU	471
DMPYH	474
DMPYHU	477
DMPYWH	480
DMPYWHU	483
DSYNC	486
DMB	488
EL_S	490
ENTER_S	492
EX	496
EXTB	499
EXTH	501
FCVT32	503
FCVT32_64	506
FCVT64	509
FCVT64_32	512
FDABS	515
FDADD	517
FDCMP	520
FDCMPF	523
FDDIV	526
FDMADD	529
FDMSUB	532
FDMUL	535
FDNEG	538
FDSQRT	540
FDSUB	542
FFS	545
FLAG	547

FLS	550
FSABS	552
FSADD	554
FSCMP	557
FSCMPF	560
FSDIV	563
FSMADD	566
FSMSUB	569
FSMUL	572
FSNEG	575
FSSQRT	577
FSSUB	579
J	582
Jcc	585
JL	588
JLcc	591
JLI_S	594
KFLAG	596
LD LDH LDW LDB LDD	599
LDI, LDI_S	604
LEAVE_S	606
LLOCK	610
LLOCKD	612
LP<cc>	614
LR	621
LSL16	623
LSL8	625
LSR	627
LSR multiple	629
LSR16	632
LSR8	634
MAC	636
MACD	639
MACDU	642
MACU	645
MAX	648
MIN	651
MOV	654
MPY, MPY_S	656
MPYD	659
MPYDU	662
MPYM MPYH	665
MPYMU, MPYHU	668
MPYU	671
MPYUW, MPYUW_S	673
MPYW, MPYW_S	675
NEG	677
NOP	679
NORM	681

NORMH, NORMW	683
NOT	686
OR	688
POP	690
POP_S	691
PREALLOC	693
PREFETCH	695
PREFETCHW	698
PUSH	700
PUSH_S	701
QMACH	703
QMACHU	706
QMPYH	709
QMPYHU	712
RCMP	715
REM	717
REMU	720
RLC	723
ROL	725
ROL8	727
ROR	729
ROR Multiple	731
ROR8	734
RRC	736
RSUB	738
RTIE	740
SBC	744
SCOND	746
SCONDD	748
SETcc	750
SETI	753
SEXB	756
SEXH, SEXW	758
SLEEP	760
SR	764
ST ST STH STB STD	766
SUB	770
SUB1	773
SUB2	775
SUB3	777
SWAP	779
SWAPE	781
SWI	783
SYNC	786
TRAP_S	789
TST	791
UNIMP_S	793
VADD2	795
VADD2H	798

VADD4H	801
VADDSUB	804
VADDSUB2H	807
VADDSUB4H	810
VMAC2H	813
VMAC2HU	815
VMPY2H	817
VMPY2HU	820
VSUB2	823
VSUB2H	826
VSUB4H	829
VSUBADD	832
VSUBADD2H	834
VSUBADD4H	837
WEVT	840
WLFC	842
XBFU	844
XOR	846
 Chapter 10	
The Host	849
10.1 The Host Interface	849
10.2 Core and Auxiliary Registers	850
10.3 Memory	850
10.4 Halting	850
10.5 Starting	851
10.6 Single Instruction Stepping	851
10.6.1 SLEEP Instruction in Single Instruction Step Mode	852
10.6.2 BRK Instruction in Single Instruction Step Mode	852
10.7 Software Breakpoints	852
 Part 2	
Memory and System Components	853
 Chapter 11	
Memory Protection Unit	855
11.1 Memory Protection Unit	855
11.1.1 Key Features	856
11.1.2 Configuration Options	856
11.1.3 Enabling the MPU	856
11.1.4 Data Organization and Addressing	856
11.1.5 Modes of Operation	860
11.1.6 Memory Protection Unit Registers	864
11.1.7 MPU Enable Register, MPU_EN	867
11.1.8 MPU Exception Cause Register, MPU_ECR	870
11.1.9 MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB31	873
11.1.10 MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP31	874
11.1.11 Memory Protection Build Configuration Register, MPU_BUILD	878
11.1.12 MPU Exceptions	879

Chapter 12

Protection Schemes	881
12.1 Stack Checking	881
12.1.1 Build Configuration Register	881
12.1.2 Enabling Stack Checking	882
12.1.3 Specifying Stack Regions	882
12.1.4 Stack-Checking Operation	883
12.1.5 Exception Information	886
12.1.6 Stack Protection Out-of-Bound Limitations	886
12.1.7 Stack Region Auxiliary Register Set	886
12.1.8 Stack Checking Exceptions	891
12.2 Coexisting Protection Schemes	892
12.3 Code Protection	893
12.3.1 Code Protection Scheme	893
12.3.2 Code Protection Behavior	893

Chapter 13

Memory Management Unit	895
13.1 MMU Introduction	895
13.2 Overview of MMU	895
13.2.1 Upper Physical Memory Address, Register, MEMSEG	897
13.2.2 TLB Page Descriptor Register for 32-Bit Virtual Address, TLBPD0	898
13.2.3 TLB Page Descriptor Register for 32-Bit Physical Address, TLBPD1	901
13.2.4 TLB Page Descriptor Register for 64-Bit Physical Address, TLBPD1_HI	904
13.2.5 TLB Index Register, TLBIndex	905
13.2.6 TLB Command Register, TLBCommand	907
13.2.7 Process Identity Register, PID	915
13.2.8 Shared Library Address Space Identifier Register, SASID0, SASID1	917
13.2.9 Scratch Data Register, SCRATCH_DATA0	919
13.3 MMU Build Configuration Register, MMU_BUILD	920

Chapter 14

ICCM	923
14.1 ICCM Auxiliary Registers	923
14.1.1 ICCM base address, AUX_ICCM	924
14.1.2 ICCM Configuration Register, ICCM_BUILD	925
14.2 Exceptions	926

Chapter 15

Instruction Cache	927
15.1 Instruction Cache Auxiliary Registers	927
15.2 Auxiliary Registers	927
15.2.1 Invalidate Instruction Cache, IC_IVIC	929
15.2.2 Instruction Cache Control Register, IC_CTRL	930
15.2.3 Lock Instruction Cache Line, IC_LIL	932
15.2.4 Invalidate Instruction-Cache Start Region, IC_IVIR	934
15.2.5 Invalidate Instruction-Cache End Region, IC_ENDR	936
15.2.6 Invalidate Instruction-Cache Line, IC_IVIL	937
15.2.7 Instruction Cache External-Access Address, IC_RAM_ADDR	938
15.2.8 Instruction-Cache Tag Access, IC_TAG	941

15.2.9 Instruction Cache Data Access, IC_DATA	945
15.2.10 Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG	948
15.2.11 Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG_HI	949
15.2.12 Instruction Cache Way-Lock Register, IC_WAYLOCK	950
15.2.13 Instruction Cache Configuration Register, I_CACHE_BUILD	951
15.3 Exceptions	952
Chapter 16	
DCCM	953
16.1 DCCM Auxiliary Registers	953
16.1.1 DCCM Base Address, AUX_DCCM	954
16.1.2 DCCM Configuration Register, DCCM_BUILD	956
16.1.3 Exceptions	959
Chapter 17	
Data Cache	961
17.1 Data Cache Auxiliary Registers	961
17.1.1 Invalidate Data Cache, DC_IVDC	963
17.1.2 Data Cache Control Register, DC_CTRL	964
17.1.3 Lock Data Cache Line, DC_LDL	966
17.1.4 Invalidate Data Line, DC_IVDL	967
17.1.5 Flush Data Cache, DC_FLSH	968
17.1.6 Flush Data Line, DC_FLDL	969
17.1.7 Data-Cache Region Start Address, DC_STARTR	970
17.1.8 Data-Cache Region End Address, DC_ENDR	971
17.1.9 Data Cache External Access Address, DC_RAM_ADDR	972
17.1.10 Data Cache Tag Access, DC_TAG	975
17.1.11 Data Cache Data Access, DC_DATA	977
17.1.12 AUX_VOLATILE	978
17.1.13 Data Cache External Access RAM Address Physical Tag Format, DC_PTAG_HI	981
17.2 Build Configuration Registers	981
17.2.1 Data Cache Configuration Register, D_CACHE_BUILD	982
17.2.2 Exceptions	983
17.3 Cache Maintenance	983
17.3.1 Cache-based Purge and Flush Operations	984
17.3.2 Region-based Flush	984
17.3.3 Region-Based Purge	984
Chapter 18	
Cluster Shared Memory	985
18.1 CSM Introduction	985
18.2 CSM Register Interface	985
18.2.1 CSM Enable Register, CSM_ENABLE	986
18.2.2 CSM ECC Control Register, CSM_ECC_CTRL	988
18.2.3 CSM SBE Counter Register, CSM_SBE_CNT	990
18.2.4 CSM ECC Initialization Register, CSM_ECC_INI	991
18.2.5 Initializing the CSM	992
18.2.6 CSM System Base Lower Address Register, CSM_BASE_SYS_ADDR_L	994
18.2.7 CSM System Base Higher Address Register, CSM_BASE_SYS_ADDR_H	995
18.2.8 CSM Build Configuration Register, CSM_BUILD	996

Chapter 19

L2 Cache	999
19.1 L2 Cache Introduction	999
19.2 L2 Cache Register Interface	999
19.3 L2 Cache Configuration Register, SLC_AUX_CACHE_CONFIG	1001
19.4 L2 Cache ECC Configuration Register, SLC_ECC_CONFIG	1003
19.5 L2 Cache Control Register, SLC_AUX_CACHE_CTRL	1005
19.5.1 Way Power Down and Power On Procedures	1008
19.6 L2 Cache Flush Register, SLC_AUX_CACHE_FLUSH	1010
19.7 L2 Cache Invalidate Register, SLC_AUX_CACHE_INV	1011
19.8 L2 Cache Way-Lock Register, SLC_AUX_CACHE_LOCK _n	1012
19.9 L2 Cache Line Lock Register, SLC_AUX_LINE_LOCK	1014
19.10 L2 Cache Line Lock High Register, SLC_AUX_LINE_LOCK1	1015
19.11 L2 Cache Line Invalidate Register, SLC_AUX_LINE_INV	1016
19.12 L2 Cache Line Invalidate High Address Register, SLC_AUX_LINE_INV1	1017
19.13 L2 Cache Line Flush Register, SLC_AUX_LINE_FLUSH	1018
19.14 L2 Cache Line Flush High Address Register, SLC_AUX_LINE_FLUSH1	1019
19.15 L2 Cache Line Operation Start Address Register, SLC_AUX_RGN_START	1020
19.16 L2 Cache Line Operation Start High Address Register, SLC_AUX_RGN_START1	1021
19.17 L2 Cache Line Operation End Address Register, SLC_AUX_RGN_END	1022
19.18 L2 Cache Line Operation End High Address Register, SLC_AUX_RGN_END1	1023
19.19 L2 Cache RAM Access Register, SLC_AUX_LINE_ADDR	1024
19.20 L2 Cache Line RAM Access High Address Register, SLC_AUX_LINE_ADDR1	1025
19.21 L2 Cache Direct Index Register, SLC_AUX_DIRECT_IDX	1026
19.22 L2 Cache TAG Register, SLC_AUX_TAG_DATA	1028
19.23 L2 Cache TAG High Address Register, SLC_AUX_TAG_DATA1	1029
19.24 L2 Cache Status Register, SLC_AUX_STATUS_DATA	1030
19.25 L2 Cache Data Register, SLC_AUX_DATA0 Register	1031
19.26 L2 Cache Data High Register, SLC_AUX_DATA1	1032
19.27 L2 Cache Data2 Register, SLC_AUX_DATA2	1033
19.28 L2 Cache Data3 Register, SLC_AUX_DATA3	1034
19.29 L2 Cache Fault Address Register, SLC_AUX_FAULT_ADDR	1035
19.30 L2 Cache Fault Address High Register, SLC_AUX_FAULT_ADDR1	1036
19.31 L2 Cache Fault Status Register, SLC_AUX_FAULT_STAT	1037
19.32 L2 Cache Performance Monitor Command Register, SLC_AUX_PM_CMD	1040
19.33 L2 Cache Performance Monitor Event Register, SLC_AUX_PM_EVENT	1042
19.34 L2 Cache Performance Counter Overflow Register, SLC_AUX_PM_OVF	1046
19.35 L2 Cache Performance Monitor Count0 Register, SLC_AUX_PM_CNT0	1047
19.36 L2 Cache Performance Monitor Count1 Register, SLC_AUX_PM_CNT1	1048
19.37 L2 Cache SBE Counter Register, SLC_AUX_ECC_SBE_CNT	1049
19.38 L2 Cache ECC Control Register, SLC_AUX_ECC_CTRL	1051
19.39 L2 Cache Build Configuration Register, SLC_BUILD	1053
19.40 Exceptions	1053

Chapter 20

Shared Memory Configuration	1055
20.1 Introduction	1055
20.2 Configuring the Shared Memory	1055
20.3 Programming the Shared Memory	1055

20.4 Programming Considerations	1055
20.5 Shared Memory Configuration Register, SM_CONFIG_REG	1058
Chapter 21	
Error Protection	1059
21.1 Auxiliary Registers	1059
21.1.1 ECC SBE Counter Register, ECC_SBE_COUNT	1060
21.1.2 Error Protection Hardware Control Register, ERP_CTRL	1064
21.1.3 Error Protection Configuration Register, ERP_BUILD	1067
21.2 Exceptions	1071
Chapter 22	
Peripheral Bus	1073
22.1 DMP Auxiliary Registers	1073
22.1.1 Peripheral Memory Region, DMP_PER_AUX	1074
22.1.2 Peripheral Region Build Configuration Register, DMP_PP_BUILD	1077
Chapter 23	
Write Buffer	1079
23.1 Introduction	1079
23.2 Configuring the Write Buffer	1079
23.3 Write Buffer Auxiliary Registers	1079
23.3.1 Write Buffer Control Register, WB_CTRL	1080
23.3.2 Write Buffer Status Register, WB_STATUS	1082
23.3.3 Write Buffer BCR Register, WB_BUILD	1083
Chapter 24	
Branch Prediction Unit	1087
24.1 Branch Predication Unit Introduction	1087
24.2 Branch Predication Unit Register Interface	1087
24.2.1 Flush and Initialize Branch Predictor Register, BPU_FLUSH	1088
24.2.2 BPU Control Register, BPU_CTRL	1089
24.2.3 Branch Cache RAM Address Register, BC_RAM_ADDR	1092
24.2.4 Branch Cache Information Register, BC_RAM_INFO	1094
24.2.5 Branch Cache Branch Target Address Register, BC_RAM_BTA	1097
24.2.6 Prediction Table Address Register, PT_RAM_ADDR	1099
24.2.7 Prediction Table DATA Register, PT_RAM_DATA	1101
24.2.8 Branch Prediction Unit Configuration Register, BPU_BUILD	1103
24.3 Debugging BPU	1104
24.3.1 Configuring BPU Debug Feature	1104
24.3.2 Enabling BPU Debug Access	1105
24.3.3 Best Practices to Follow When Debugging BPU	1105
24.3.4 Methods to Access the BPU Auxiliary Registers	1105
24.3.5 What Data Can be Accessed	1106
24.3.6 How to Dump Branch Cache and Prediction Table Data	1106
24.3.7 Usage Examples	1106
Chapter 25	
Loop Buffer	1113
25.1 Introduction	1113

25.2	Configuring the Loop Buffer	1113
25.3	Programming the Loop Buffer	1113
25.3.1	Loop Buffer Control, LPB_CTRL	1114
25.3.2	Loop Buffer Configuration Register, LPB_BUILD	1115
Chapter 26		
External Host Debugging		1117
26.1	External Host Debugging Introduction	1117
26.2	Host Debug Register Interface	1117
26.2.1	Debug Register, DEBUG	1118
26.2.2	Interrupt and Register Bank Debug Register, DEBUGI	1122
Chapter 27		
Debug Features		1125
27.1	Debug Introduction	1125
27.2	Actionpoints	1125
27.2.1	Actionpoint Auxiliary Registers	1125
27.2.2	Actionpoint Extensions to Debug Register, DEBUG, 0x05	1129
27.2.3	Actionpoints Configuration Register, AP_BUILD	1130
27.2.4	Actionpoint Match Value, AP_AMV0	1131
27.2.5	Actionpoint Match Mask, AP_AMM0	1132
27.2.6	Actionpoint Control, AP_AC0	1133
27.2.7	Actionpoint Match Value, AP_AMV1	1138
27.2.8	Actionpoint Match Mask, AP_AMM1	1139
27.2.9	Actionpoint Control, AP_AC1	1140
27.2.10	Actionpoint Match Value, AP_AMV2	1141
27.2.11	Actionpoint Match Mask, AP_AMM2	1142
27.2.12	Actionpoint Control, AP_AC2	1143
27.2.13	Actionpoint Match Value, AP_AMV3	1144
27.2.14	Actionpoint Match Mask, AP_AMM3	1145
27.2.15	Actionpoint Control, AP_AC3	1146
27.2.16	Actionpoint Match Value, AP_AMV4	1147
27.2.17	Actionpoint Match Mask, AP_AMM4	1148
27.2.18	Actionpoint Control, AP_AC4	1149
27.2.19	Actionpoint Match Value, AP_AMV5	1150
27.2.20	Actionpoint Match Mask, AP_AMM5	1151
27.2.21	Actionpoint Control, AP_AC5	1152
27.2.22	Actionpoint Match Value, AP_AMV6	1153
27.2.23	Actionpoint Match Mask, AP_AMM6	1154
27.2.24	Actionpoint Control, AP_AC6	1155
27.2.25	Actionpoint Match Value, AP_AMV7	1156
27.2.26	Actionpoint Match Mask, AP_AMM7	1157
27.2.27	Actionpoint Control, AP_AC7	1158
27.2.28	Actionpoint Isolation Enable Register, AP_ISO_EN	1159
27.2.29	Watchpoint Program Counter, AP_WP_PC	1160
27.3	SmaRT	1160
27.3.1	Features	1160
27.3.2	Overview of SmaRT Operation	1161
27.3.3	Auxiliary Registers	1162

27.3.4 SMART_BUILD Configuration Register, SMART_BUILD	1163
27.3.5 Smart Control Register, SMART_CONTROL	1164
27.3.6 Smart Data Register, SMART_DATA	1166
27.3.7 SRC_ADDR Value, Index 0	1166
27.3.8 DEST_ADDR Value, Index 1	1167
27.3.9 FLAGS Value, Index 2	1167
27.4 Debug Interface	1168
27.4.1 ARC Debug-Access Registers	1168
27.4.2 Debug Register, DB_STATUS	1170
27.4.3 Debug Register, DB_CMD	1171
27.4.4 Debug Register, DB_ADDR	1172
27.4.5 Debug Register, DB_DATA	1173
27.4.6 Debug Register, DB_RESET	1174
27.4.7 Initiating a Basic Debug Operation	1174
Chapter 28	
Performance Counters	1177
28.1 Performance Counters Introduction	1177
28.2 Performance Counters Registers	1178
28.2.1 Countable Conditions Index Register, CC_INDEX	1180
28.2.2 Countable Conditions Name0 Register, CC_NAME0	1181
28.2.3 Countable Conditions Name1 Register, CC_NAME1	1182
28.2.4 Count-Value Registers, PCT_COUNTL	1183
28.2.5 Count-Value Registers, PCT_COUNTH	1184
28.2.6 Snapshot-Value Registers, PCT_SNAPL	1185
28.2.7 Snapshot-Value Registers, PCT_SNAPH	1186
28.2.8 Configuration Register, PCT_CONFIG	1187
28.2.9 Control Register, PCT_CONTROL	1188
28.2.10 Index-Select Register, PCT_INDEX	1190
28.2.11 Address-Range Registers, PCT_RANGEL	1191
28.2.12 Address-Range Registers, PCT_RANGEH	1192
28.2.13 User-Flag Register, PCT_UFLAGS	1193
28.2.14 Interrupt Trigger Value, PCT_INT_CNTL	1195
28.2.15 Interrupt Trigger Value, PCT_INT_CNTH	1196
28.2.16 Interrupt Control Value, PCT_INT_CTRL	1197
28.2.17 Interrupt Active, PCT_INT_ACT	1198
28.2.18 Performance Counter Build-Configuration Register, PCT_BUILD	1199
28.2.19 Countable Conditions Build Configuration Register, CC_BUILD	1200
Chapter 29	
Processor Timers	1201
29.1 Timers	1201
29.2 Auxiliary Timer Registers	1202
29.2.1 Timer 0 Count Register, COUNT0	1204
29.2.2 Timer 0 Control Register, CONTROL0	1205
29.2.3 Timer 0 Limit Register, LIMIT0	1207
29.2.4 Timer 1 Count Register, COUNT1	1208
29.2.5 Timer 1 Control Register, CONTROL1	1209
29.2.6 Timer 1 Limit Register, LIMIT1	1210

29.2.7	RTC Control Register, AUX_RTC_CTRL	1211
29.2.8	RTC Count Low Register, AUX_RTC_LOW	1213
29.2.9	RTC Count High Register, AUX_RTC_HIGH	1214
29.2.10	Timers Configuration Register, TIMER_BUILD	1215
Chapter 30		
Multi-core Cluster		1217
30.1	Multi-core Cluster Introduction	1217
30.2	Cluster Auxiliary and Build Configuration Registers	1218
30.2.1	CLUSTER ID Register, CLUSTER_ID	1219
30.2.2	Global Clock Disable Register, GLOBAL_CLK_GATE_DIS	1220
30.2.3	Cluster VCCM QoS Register, CLUSTER_VCCM_QoS	1222
30.2.4	Cluster NMI Status Register, CC_NMI_STATUS	1224
30.2.5	Cluster NMI Cause Register, CC_NMI_CAUSE	1226
30.2.6	CLUSTER Build Configuration Register, CLUSTER_BUILD	1230
Chapter 31		
Shared Coherency Unit		1233
31.1	Auxiliary Registers	1233
31.1.1	I/O Coherency Enable Register, BIU_IO_0_COH_ENABLE	1234
31.1.2	I/O Coherency Partial Write Register, BIU_IO_0_COH_PARTIAL	1235
31.1.3	I/O Coherency Address Aperture Base Address, BIU_IO_0_COH_AP0_BASE	1236
31.1.4	I/O Coherency Aperture Size, BIU_IO_0_COH_AP0_SIZE	1237
31.1.5	Bus Interface Unit Build Configuration Register, BIU_BUILD	1238
Chapter 32		
Power Management Features		1241
32.1	Introduction to Power Management Features	1241
32.2	PDM and DVFS Register Interface	1241
32.2.1	Core Power Status Register, PDM_PSTAT	1243
32.2.2	ARC Trace Power Status Register, RTT_PDM_PSTAT	1245
32.2.3	Power Down Register, PDM_PMODE	1246
32.2.4	Core Snoop Indicator Control Register, PDM_SCU_DIS_IND	1247
32.2.5	Disable SCU Transaction	1247
32.2.6	Enable SCU Transaction	1247
32.2.7	Power Down Cluster Register, PDM_CC_PMODE	1248
32.2.8	Cluster Power Status Register, PDM_CC_PSTAT	1249
32.2.9	Power Down Cluster Shared Memory Register, PDM_CSM_PMODE	1252
32.2.10	Cluster CSM Power Status Register, PDM_CSM_PSTAT	1253
32.2.11	Power Down L2 Cache Register, PDM_SLC_PMODE	1255
32.2.12	Cluster L2 Cache Power Status Register, PDM_SLC_PSTAT	1256
32.2.13	Partial Power-Down Cluster Shared Memory Register, PDM_CSM_P_PMODE	1257
32.2.14	Partial Power-Down Cluster Shared Memory Register, PDM_CSM_P_PSTAT	1259
32.2.15	Partial Power-Down L2 Cache Register, PDM_SLC_P_PMODE	1262
32.2.16	Partial Power-Down L2 Cache Register, PDM_SLC_P_PSTAT	1264
32.2.17	Shared Memory Power Down Register, PDM_CSM_SM_P_CFG	1267
32.2.18	Cluster Performance Register, DVFS_CLUSTER_PL	1269
32.3	Execution Rate Interface	1269
32.3.1	Execution Rate Control Register, EXEC_CTRL	1270
32.4	Build Configuration Registers	1270

32.4.1 Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD .1271

Chapter 33

ARConnect	1273
33.1 ARConnect Introduction	1273
33.2 ARConnect Register Interface	1273
33.2.1 ARConnect Command Register, CONNECT_CMD	1275
33.2.2 ARConnect Write Data Register, CONNECT_WDATA	1276
33.2.3 ARConnect Read Data Register, CONNECT_READBACK	1277
33.2.4 ARConnect Build Configuration Register, CONNECT_SYSTEM_BUILD	1278
33.2.5 Inter-core Semaphore Unit BCR, CONNECT_SEMA_BUILD	1280
33.2.6 Inter-core Message Unit BCR, CONNECT_MESSAGE_BUILD	1281
33.2.7 Power Management Unit BCR, CONNECT_PMU_BUILD	1283
33.2.8 Interrupt Distribution Unit BCR, CONNECT_IDU_BUILD	1284
33.2.9 ARConnect Global Free Running Counter BCR, CONNECT_GFRC_BUILD	1285
33.2.10 Inter-Core Interrupt Vector Control Unit BCR, CONNECT_IVC_BUILD	1286
33.2.11 Inter-Core Interrupt Unit BCR, CONNECT_ICI_BUILD	1287
33.2.12 Inter-Core Debug Unit BCR, CONNECT_ICD_BUILD	1288
33.2.13 ARConnect Slave Interface Unit BCR, CONNECT_ASI_BUILD	1289
33.2.14 ARConnect PDM BCR, CONNECT_PDM_BUILD	1290
33.3 Programming Restrictions	1290
33.3.1 Atomic Operation	1290
33.3.2 Preventing Conflicts	1290

Chapter 34

Cluster DMA Controller	1293
34.1 High-Level Architecture	1293
34.2 Use Cases	1296
34.3 Configuration Options	1296
34.4 Programming the Cluster DMA using Auxiliary Registers	1296
34.4.1 DMA Client Control Register, DMA_C_CTRL_AUX	1298
34.4.2 DMA Client Channel Select Register, DMA_C_CHAN_AUX	1299
34.4.3 DMA Client Source Address Register, DMA_C_SRC_AUX	1300
34.4.4 DMA Client Source High Address Register, DMA_C_SRC_HI_AUX	1301
34.4.5 DMA Client Destination Address Register, DMA_C_DST_AUX	1302
34.4.6 DMA Client Destination High Address Register, DMA_C_DST_HI_AUX	1303
34.4.7 DMA Client Attributes Register, DMA_C_ATTR_AUX	1304
34.4.8 DMA Client Length Register, DMA_C_LEN_AUX	1306
34.4.9 DMA Client Handle Register, DMA_C_HANDLE_AUX	1307
34.4.10 DMA Event Status Register, DMA_C_EVSTAT_AUX	1308
34.4.11 DMA Client Clear Event Status Register, DMA_C_EVSTAT_CLR_AUX	1309
34.4.12 DMA Client Status Register, DMA_C_STATUS_AUX	1310
34.4.13 DMA Client Interrupt Status Register, DMA_C_INTSTAT_AUX	1311
34.4.14 DMA Client Clear Interrupt Status Register, DMA_C_INTSTAT_CLR_AUX	1313
34.4.15 DMA Client Error Handle Register, DMA_C_ERRHANDLE_AUX	1314
34.4.16 DMA Server Control Register, DMA_S_CTRL_AUX	1315
34.4.17 DMA Server Done Status Register, DMA_S_DONESTATD_AUX	1316
34.4.18 DMA Server Clear Done Status Register, DMA_S_DONESTATD_CLR_AUX	1317
34.4.19 DMA Server Channel Tail Pointer Register, DMA_S_TAILC_AUX	1318

34.4.20 DMA Server Channel Middle Pointer Register, DMA_S_MIDC_AUX	1319
34.4.21 DMA Server Channel Head Pointer Register, DMA_S_HEADC_AUX	1320
34.4.22 DMA Server Channel Base Register, DMA_S_BASEC_AUX	1321
34.4.23 DMA Server Channel Last Register, DMA_S_LASTC_AUX	1322
34.4.24 DMA Channel Priority Register, DMA_S_PRIORC_AUX	1323
34.4.25 DMA Channel Control Register, DMA_S_STATC_AUX	1324
34.5 Steps to Initiate a DMA Transfer	1324
34.5.1 DMA Build Configuration Register, DMA_BUILD	1327
Chapter 35	
Floating-Point Unit	1329
35.1 Core Register Set	1329
35.2 Extension Core Registers	1330
35.3 FPU Auxiliary Registers	1331
35.3.1 Floating-Point Unit Control Register, FPU_CTRL	1332
35.3.2 Floating-Point Unit Status Register, FPU_STATUS	1334
35.4 Build Configuration Registers	1337
35.4.1 Floating-Point Unit Build Register, FPU_BUILD	1338
Chapter 36	
User Auxiliary Register Interface	1341
Chapter 37	
ROM Patching Unit	1343
37.1 Introduction	1343
37.2 Patch Entry	1344
37.3 Patch Breakpoints	1344
37.4 Patch Jump	1344
37.4.1 BPU	1345
37.4.2 Delay Slots	1345
37.5 Exceptions	1345
37.6 Build-Time Decisions	1345
37.7 Programming the RPU	1345
37.8 Programming Constraints	1346
37.9 Auxiliary Registers	1347
37.9.1 RPU Command Register, RPU_CMND	1348
37.9.2 RPU Address Register, RPU_ADDR	1349
37.9.3 RPU DATA Register, RPU_DATA	1351
37.9.4 RPU Enable Register, RPU_ENABLE	1352
37.9.5 RPU MODE Register, RPU_MODE	1353
37.9.6 RPU Status Register, RPU_STATUS	1354
37.9.7 ROM Base Address Register, RPU_ROM_BASE	1355
37.9.8 RPU Patch Table Base Address Register, RPU_PATCH_TABLE_BASE	1356
37.9.9 RPU Patch Entry Register, PatchTable[n].Address	1357
37.9.10 RPU Build Configuration Register, RPU_BUILD	1358
Part 3	
FastMath Extension Pack	1359
Chapter 38	

FastMath	1361
38.1 FastMath Data Organization and Addressing	1361
38.1.1 Fractional Data Formats	1361
38.1.2 16-Bit Signed Fractions	1362
38.1.3 32-Bit Signed Fraction (1Q31)	1362
38.2 FastMath Register Set Extensions	1362
38.2.1 Extension Auxiliary Registers	1362
38.2.2 FMP Control Register, FMP_CTRL	1363
38.2.3 Build Configuration Registers	1363
38.2.4 Extension Core Registers	1364
38.3 FastMath Extension Instruction Details	1364
38.3.1 Semantics of FastMath Arithmetic Operations	1365
38.3.2 FastMath fixed-point type declarations	1365
38.3.3 Multiplication of extended-precision 37-bit signed integer values	1367
38.3.4 Rounding of fractional values	1367
38.3.5 Bit-extraction and rounding functions	1368
38.3.6 Support functions for trigonometric operations	1369
38.3.7 Saturating result values to a defined range	1370
38.3.8 Computation of LOG2 (1Q31)	1370
38.3.9 Computation of EXP2 (1Q31)	1372
38.3.10 Computation of COS (1Q31)	1375
38.3.11 Computation of SIN (1Q31)	1376
38.3.12 Computation of ATAN (1Q31)	1377
38.3.13 Computation of SQRT (1Q31)	1378
38.3.14 1Q31 and 1Q15 Versions of SQRT, LOG2, EXP2, SIN, COS and ATAN	1379
FMP_ADDS	1381
FMP_RNDH	1383
FMP_SATH	1385
FMP_DIVF	1387
FMP_DIVF15	1389
FMP_RECIP	1391
FMP_RECIP15	1393
FMP_SQRTF	1395
FMP_SQRTF15	1397
FMP_COS	1399
FMP_COS15	1401
FMP_SIN	1403
FMP_SIN15	1405
FMP_ATAN	1407
FMP_ATAN15	1409
FMP_EXP2	1411
FMP_EXP215	1413
FMP_LOG2	1415
FMP_LOG215	1417

Part 4

ARCV2 Scalar DSP Extensions 1419

Chapter 38

Scalar DSP Introduction	1421
38.1 Key Features of the ARCV2 ISA DSP Extensions	1421
38.2 Programmer's Model	1422
38.3 Configurability	1422
Chapter 39	
Data Organization and Addressing	1423
39.1 DSP Data Formats	1423
39.1.1 8-Bit Signed Integer Vector	1423
39.1.2 16-Bit Signed Integer	1423
39.1.3 16-Bit Unsigned Integer	1423
39.1.4 16-Bit Signed Fractional	1424
39.1.5 16-Bit Signed Integer Vector	1424
39.1.6 16-Bit Unsigned Integer Vector	1424
39.1.7 16-Bit Signed Fractional Vector	1424
39.1.8 Complex 16+16 Bit Signed Fractional Data	1425
39.1.9 32-Bit Signed Integer	1425
39.1.10 32-Bit Unsigned Integer	1425
39.1.11 32-Bit Signed Fractional	1425
39.1.12 Accumulators	1426
39.1.13 Accumulator Saturation	1428
39.1.14 Accumulator Operations	1428
Chapter 40	
DSP Auxiliary Interface	1431
40.1 DSP Auxiliary Registers	1431
40.1.1 Accumulator Low Register, ACC0_LO	1432
40.1.2 Low Accumulator Guard and Status Register, ACC0_GLO	1433
40.1.3 Accumulator High Register, ACC0_HI	1434
40.1.4 High Accumulator Guard and Status Register, ACC0_GHI	1435
40.1.5 DSP Butterfly Instructions Data Register, DSP_BFLY0	1436
40.1.6 DSP FFT Control Register, DSP_FFT_CTRL	1437
40.1.7 DSP Control Register, DSP_CTRL	1438
40.2 Build Configuration Registers	1439
40.2.1 DSP Build Configuration Register, DSP_BUILD	1440
Chapter 41	
Address Generation Unit Auxiliary Interface	1441
41.1 Address Generation	1441
41.2 Limitations	1443
41.3 AGU Auxiliary Register Set	1444
41.3.1 AGU Address Pointer Registers, AGU_AUX_APx	1445
41.3.2 AGU Offset Registers, AGU_AUX_OSx	1448
41.3.3 AGU Modifier Registers, AGU_AUX_MODx	1449
41.3.4 AGU Build Configuration Register, AGU_BUILD	1452
Chapter 42	
Scalar DSP Instruction Set Details	1453
42.1 Notations Used in the DSP Instructions	1453
42.2 List of Instructions	1453

ABSS	1461
ABSSH	1463
ADCS	1465
ADDS	1467
ASLACC	1469
ASLS	1473
ASLSACC	1475
ASRS	1480
ASRSR	1482
CBFLYHF0R	1484
CBFLYHF1R	1488
CMACCHFR	1492
CMACCHNFR	1496
CMACHFR	1499
CMACHNFR	1503
CMPYCHFR	1506
CMPYCHNFR	1510
CMPYHFR	1513
CMPYHFMR	1517
CMPYHNFR	1521
DIV	1524
DIVF	1527
DIVU	1530
DMACH	1533
DMACHBL	1536
DMACHBM	1539
DMACHF	1542
DMACHFR	1545
DMACHU	1548
DMACWH	1551
DMACWHF	1554
DMACWHU	1557
DMPYH	1560
DMPYHBL	1562
DMPYHBM	1564
DMPYHF	1566
DMPYHFR	1569
DMPYHU	1572
DMPYWH	1574
DMPYHWF	1577
DMPYWHF	1580
DMPYWHU	1583
FLAGACC	1586
GETACC	1588
MAC	1593
MACD	1596
MACDF	1599
MACDU	1602
MACF	1605

MACFR	1608
MACU	1611
MACWHFM	1614
MACWHFMR	1617
MACWHL	1620
MACWHFL	1623
MACWHFLR	1626
MACWHKL	1629
MACWHKUL	1632
MACWHUL	1635
MODIF	1638
MODAPP	1642
MPY	1644
MPY_S	1647
MPYD	1649
MPYDF	1652
MPYDU	1655
MPYF	1658
MPYFR	1661
MPYM MPYH	1664
MPYMU MPYHU	1666
MPYU	1668
MPYUW	1670
MPYUW_S	1672
MPYW	1674
MPYW_S	1676
MPYWHFL	1678
MPYWHFLR	1681
MPYWHFM	1684
MPYWHFMR	1687
MPYWHL	1690
MPYWHUL	1693
MPYWHKL	1696
MPYWHKUL	1699
MSUBDF	1702
MSUBF	1705
MSUBFR	1708
MSUBWHFL	1711
MSUBWHFLR	1714
MSUBWHFM	1717
MSUBWHFMR	1720
NEGS	1723
NEGSH	1725
NORMACC	1727
QMACH	1731
QMACHF	1734
QMACHU	1737
QMPYH	1740
QMPYHF	1743

QMPYHU	1746
REM	1749
REMU	1752
RNDH	1755
SATH	1757
SATF	1759
SBCS	1761
SETACC	1763
SQRT	1768
SQRTF	1770
SUBS	1772
VABS2H	1774
VABSS2H	1776
VADD2H	1778
VADD4B	1780
VADD52	1783
VADD2	1785
VADD4H	1787
VADDSUB	1789
VADDSUB2H	1791
VADDSUB4H	1793
VADD52H	1796
VADD54H	1798
VADDSUB2H	1800
VADDSUBS	1802
VADDSUBS2H	1804
VADDSUBS4H	1807
VALGN2H	1810
VASL2H	1812
VASLS2H	1814
VASR2H	1817
VASRS2H	1819
VASRSR2H	1822
VEXT2BHL	1825
VEXT2BHLF	1827
VEXT2BHM	1829
VEXT2BHMF	1831
VLSR2H	1833
VMAC2H	1835
VMAC2HF	1837
VMAC2HFR	1841
VMAC2HNFR	1845
VMAC2HU	1848
VMAX2H	1850
VMIN2H	1852
VMPY2H	1854
VMPY2HF	1856
VMPY2HFR	1860
VMPY2HU	1864

VMPY2HWF	1867
VMSUB2HF	1870
VMSUB2HFR	1874
VMSUB2HNFR	1878
VNEG2H	1881
VNEGS2H	1883
VNORM2H	1885
VPACK2HL	1887
VPACK2HM	1889
VPACK2HBL	1891
VPACK2HBLF	1893
VPACK2HBM	1895
VPACK2HBMF	1897
VPERM	1899
VREP2HL	1902
VREP2HM	1904
VSEXT2BHL	1906
VSEXT2BHM	1908
VSUB2	1910
VSUB2H	1912
VSUB4H	1914
VSUBADD	1916
VSUBADD2H	1918
VSUBADD4H	1920
VSUB2H	1922
VSUBS2	1924
VSUBS2H	1926
VSUBS4H	1928
VSUB4B	1930
VSUBADDS	1932
VSUBADDS2H	1934
VSUBADDS4H	1936

Part 5

Appendices 1939

Appendix A

ISA Version Differences	1941

Appendix B

Implementation-dependent Behavior	1949
B.1 EM Exception Handling	1949
B.1.1 ECR User-mode Setting in the ARC EM Family of Cores	1949
B.2 Cause Codes for Memory Accesses and Data/Address Errors	1949
B.3 Exception Handling	1950
B.3.1 ECR User-mode Setting for the ARC HS/EV/VPX5 Family of Cores	1950
B.4 Result on Overflow from Integer Division	1950

Appendix C

ARConnect Commands Details	1951

C.1	CMD_CHECK_CORE_ID	1951
C.2	CMD_INTRPT_GENERATE_IRQ	1952
C.3	CMD_INTRPT_GENERATE_ACK	1953
C.4	CMD_INTRPT_READ_STATUS	1954
C.5	CMD_INTRPT_CHECK_SOURCE	1955
C.6	CMD_INTRPT_GENERATE_ACK_BIT_MASK	1956
C.7	CMD_INTRPT_EXT_MODE	1957
C.8	CMD_INTRPT_SET_PULSE_CNT	1958
C.9	CMD_INTRPT_READ_PULSE_CNT	1958
C.10	CMD_SEMA_CLAIM_AND_READ	1959
C.11	CMD_SEMA_RELEASE	1960
C.12	CMD_SEMA_FORCE_RELEASE	1960
C.13	CMD_MSG_SRAM_SET_ADDR	1961
C.14	CMD_MSG_SRAM_READ_ADDR	1961
C.15	CMD_MSG_SRAM_SET_ADDR_OFFSET	1962
C.16	CMD_MSG_SRAM_READ_ADDR_OFFSET	1963
C.17	CMD_MSG_SRAM_WRITE	1963
C.18	CMD_MSG_SRAM_WRITE_INC	1964
C.19	CMD_MSG_SRAM_WRITE_IMM	1965
C.20	CMD_MSG_SRAM_READ	1965
C.21	CMD_MSG_SRAM_READ_INC	1966
C.22	CMD_MSG_SRAM_READ_IMM	1967
C.23	CMD_MSG_SET_ECC_CTRL	1967
C.24	CMD_MSG_READ_ECC_CTRL	1969
C.25	CMD_MSG_READ_ECC_SBE_CNT	1969
C.26	CMD_MSG_SRAM_CLEAR_ECC_SBE_CNT	1970
C.27	CMD_DEBUG_RESET	1971
C.28	CMD_DEBUG_HALT	1972
C.29	CMD_DEBUG_RUN	1972
C.30	CMD_DEBUG_SET_MASK	1973
C.31	CMD_DEBUG_READ_MASK	1974
C.32	CMD_DEBUG_SET_SELECT	1975
C.33	CMD_DEBUG_READ_SELECT	1975
C.34	CMD_DEBUG_READ_EN	1976
C.35	CMD_DEBUG_READ_CMD	1977
C.36	CMD_DEBUG_READ_CORE	1977
C.37	CMD_GFRC_CLEAR	1978
C.38	CMD_GFRC_READ_LO	1979
C.39	CMD_GFRC_READ_HI	1979
C.40	CMD_GFRC_ENABLE	1980
C.41	CMD_GFRC_DISABLE	1980
C.42	CMD_GFRC_READ_DISABLE	1981
C.43	CMD_GFRC_SET_CORE	1981
C.44	CMD_GFRC_READ_CORE	1983
C.45	CMD_GFRC_READ_HALT	1983
C.46	CMD_GFRC_CLK_ENABLE	1984
C.47	CMD_GFRC_CLK_DISABLE	1984
C.48	CMD_GFRC_READ_CLK_STATUS	1985
C.49	CMD_PDM_SET_PM	1985

C.50	CMD_PDM_READ_PSTATUS	1986
C.51	CMD_IVC_SET	1987
C.52	CMD_IVC_READ	1988
C.53	CMD_PMU_SET_PUCNT	1989
C.54	CMD_PMU_READ_PUCNT	1990
C.55	CMD_PMU_SET_RSTCNT	1990
C.56	CMD_PMU_READ_RSTCNT	1991
C.57	CMD_PMU_SET_PDCNT	1991
C.58	CMD_PMU_READ_PDCNT	1992
C.59	CMD_IDU_ENABLE	1993
C.60	CMD_IDU_DISABLE	1993
C.61	CMD_IDU_READ_ENABLE	1994
C.62	CMD_IDU_SET_MODE	1994
C.63	CMD_IDU_READ_MODE	1995
C.64	CMD_IDU_SET_DEST	1996
C.65	CMD_IDU_READ_DEST	1997
C.66	CMD_IDU_GEN_CIRQ	1997
C.67	CMD_IDU_ACK_CIRQ	1998
C.68	CMD_IDU_CHECK_STATUS	1999
C.69	CMD_IDU_CHECK_SOURCE	2000
C.70	CMD_IDU_SET_MASK	2002
C.71	CMD_IDU_READ_MASK	2003
C.72	CMD_IDU_CHECK_FIRST	2003
C.73	Examples: ARConnect ICI	2005
C.73.1	Scenario 1	2005
C.73.2	Scenario2	2006
C.74	Examples: ARConnect Semaphore Unit	2008
C.74.1	Scenario 1	2008
C.74.2	Scenario 2	2008
C.75	Examples: ICM	2009
C.75.1	Scenario 1	2009
C.76	Examples: ICD	2010
C.76.1	Conditionally Halt Cores Scenario	2010
C.76.2	Selectively Halt Cores Scenario	2010
C.76.3	Selectively Run Cores Scenario	2010
C.76.4	Selectively Reset Cores Scenario	2011
C.77	Examples: GFRC	2012
C.78	Examples: IDU	2012
C.79	Examples: IVC	2014
Index		2015

List of Figures

Figure 3-1 Address Space Model	109
Figure 3-2 64-bit Element As a 64-bit Operand	110
Figure 3-3 32-bit Elements in a 64-bit Operand	110
Figure 3-4 16-bit Elements in a 64-bit Operand	110
Figure 3-5 Register Pair Holding Operands in Little-endian Mode	110
Figure 3-6 Register Pair Holding Operands in Big-endian Mode	111
Figure 3-7 32-Bit Elements in a 32-bit Operand	112
Figure 3-8 16-Bit Elements in a 32-bit Operand	112
Figure 3-9 Expansion of a u6 Literal to a 32-Bit Word (w-shimm)	112
Figure 3-10 Expansion of an s12 Literal to a 32-Bit Word (w-shimm)	112
Figure 3-11 Expansion of a u6 Literal to Half-Word (16-bit, hw-shimm)	112
Figure 3-12 Expansion of an s12 Literal to Half-Word (16-bit, hw-shimm)	112
Figure 3-13 A Half-Word (16-bit) Short Limm Value Replicated to form a 64-Bit Operand	112
Figure 3-14 A word (32-bit) Short-limm or Limm value Replicated to form a 64-Bit Operand	113
Figure 3-15 Two 16-Bit Vectors with Expanded u6 Operands	113
Figure 3-16 Two 16-Bit Vectors with Expanded s12 Operands	113
Figure 3-17 Four 8-Bit Vectors with Expanded u6 Operands	113
Figure 3-18 Four 16-Bit Vectors with Expanded s12 Operands	113
Figure 3-19 Expansion of a w6 in STD	113
Figure 3-20 Expansion of a limm in STD	113
Figure 3-21 Half-Precision Floating-Point Data	114
Figure 3-22 Single-Precision Floating-Point Data	114
Figure 3-23 Double-Precision Floating-Point Data	115

Figure 3-24 64-bit Register Data in Byte-Wide Memory, Little-Endian	117
Figure 3-25 64-bit Register Data in Byte-Wide Memory, Big-Endian	117
Figure 3-26 Register containing 32-bit Data	118
Figure 3-27 32-bit Register Data in Byte-Wide Memory, Little-Endian	118
Figure 3-28 32-bit Register Data in Byte-Wide Memory, Big-Endian	118
Figure 3-29 Register Containing 16-Bit Data.	119
Figure 3-30 16-bit Register Data in Byte-Wide Memory, Little-Endian	119
Figure 3-31 16-bit Register Data in Byte-Wide Memory, Big-Endian	119
Figure 3-32 Register Containing 8-Bit Data.	119
Figure 3-33 8-bit Register Data in Byte-Wide Memory	119
Figure 3-34 Register Containing 1-Bit Data.	120
Figure 3-35 Half-word Numbering of a 32-Bit Instruction or Long-immediate Data Item	120
Figure 3-36 Little-endian Offset of each Byte in a 32-Bit Instruction or Immediate	120
Figure 3-37 Big-endian Offset of each Byte in a 32-Bit Instruction or Immediate	120
Figure 3-38 Little-endian Memory Offset of each Byte in a 16-Bit Instruction	121
Figure 3-39 Big-endian Memory Offset of each Byte in a 16-bit Instruction	121
Figure 4-1 Core Register Map Summary	125
Figure 4-2 PCL Register	133
Figure 4-3 ACCH for Double-Precision FPU operations	134
Figure 4-4 ACCL for Double-Precision FPU operations	134
Figure 4-5 ACCL for Single-Precision FPU Operations	134
Figure 4-6 LP_START Register (PC_SIZE == 32)	139
Figure 4-7 LP_START Register (PC_SIZE < 32)	139
Figure 4-8 LP_END Register	140
Figure 4-9 LP_END Register (PC_SIZE < 32)	140
Figure 4-10 IDENTITY Register	141
Figure 4-11 PC Register Addressing Full 32-bit Address Space	143
Figure 4-12 PC Register Addressing a Reduced Address Space (PC_SIZE == 32)	143
Figure 4-13 STATUS32 Register	145

Figure 4-14 STATUS32_P0 Register	151
Figure 4-15 AUX_USER_SP Register	152
Figure 4-16 AUX_IRQ_CTRL Register.	153
Figure 4-17 INT_VECTOR_BASE Register (PC_SIZE == 32).	154
Figure 4-18 INT_VECTOR_BASE Register (PC_SIZE < 32)	154
Figure 4-19 AUX_IRQ_ACT Register.	155
Figure 4-20 IRQ_PRIORITY_PENDING Register	156
Figure 4-21 AUX_IRQ_HINT Register.	157
Figure 4-22 IRQ_PRIORITY Register	158
Figure 4-23 DATA_MEM_ATTR	160
Figure 4-24 JLI_BASE Register when PC_SIZE == 32	162
Figure 4-25 JLI_BASE Register when PC_SIZE < 32	162
Figure 4-26 LDI_BASE Register when ADDR_SIZE == 32.	163
Figure 4-27 LDI_BASE Register when ADDR_SIZE < 32	163
Figure 4-28 EI_BASE Register when PC_SIZE == 32.	164
Figure 4-29 EI_BASE Register when PC_SIZE < 32.	164
Figure 4-30 ERET Register when PC_SIZE == 32	165
Figure 4-31 ERET Register when PC_SIZE < 32.	165
Figure 4-32 ERBTA Register when PC_SIZE == 32	166
Figure 4-33 ERBTA Register when PC_SIZE < 32	166
Figure 4-34 ERSTATUS Register.	167
Figure 4-35 ECR Register	168
Figure 4-36 EFA Register	170
Figure 4-37 ICAUSE Register	172
Figure 4-38 IRQ_SELECT Register	173
Figure 4-39 IRQ_ENABLE Register	174
Figure 4-40 IRQ_TRIGGER Register	175
Figure 4-41 IRQ_STATUS Register.	176
Figure 4-42 XPU Register	177

Figure 4-43 BTA Register (PC_SIZE == 32).....	179
Figure 4-44 BTA Register (PC_SIZE < 32).....	179
Figure 4-45 IRQ_PULSE_CANCEL Register	181
Figure 4-46 IRQ_PENDING Register.....	182
Figure 4-47 User Extension Flags Register, XFLAGS	183
Figure 4-48 BCR_VER Register.....	185
Figure 4-49 BTA_LINK_BUILD Register.....	186
Figure 4-50 VECBASE_AC_BUILD Register	187
Figure 4-51 RF_BUILD Register	189
Figure 4-52 MULTIPLY_BUILD Register	191
Figure 4-53 SWAP_BUILD Register.....	192
Figure 4-54 NORM_BUILD Register	193
Figure 4-55 MINMAX_BUILD Register.....	194
Figure 4-56 BARREL_BUILD Register.....	195
Figure 4-57 ISA_CONFIG Build Register	196
Figure 4-58 IRQ_BUILD Register	199
Figure 4-59 ISA_PROFILE Register	201
Figure 4-60 MICRO_ARCH_BUILD Register.....	203
Figure 5-1 Fast Interrupt Entry	217
Figure 5-2 Fast Interrupt Exit	218
Figure 5-3 Determine Active Interrupts	226
Figure 6-1 Long Immediate source data value.....	264
Figure 7-1 Branch Conditionally Format.....	320
Figure 7-2 Branch Unconditional Far Format.....	320
Figure 7-3 Branch on Compare Register-register Format.....	321
Figure 7-4 Branch on Compare Register-Immediate Format.....	322
Figure 7-5 Branch and Link Conditionally Format	323
Figure 7-6 Branch and Link Unconditional Far Format	324
Figure 7-7 Load Register with Offset Format.....	325

Figure 7-8 Store Register with Offset Format	326
Figure 7-9 Store Immediate with Offset Format.	326
Figure 7-10 General Operations Register-Register Format.	328
Figure 7-11 General Operations Register with Unsigned 16-bit Immediate Format	329
Figure 7-12 General Operations Register with Signed 12-bit Immediate Format	329
Figure 7-13 General Operations Conditional Register Format	329
Figure 7-14 General Operations Conditional Register with Unsigned 6-bit Immediate	330
Figure 7-15 Load Register-Register Format	333
Figure 7-16 Extension ALU Operation, register-register	340
Figure 7-17 Extension ALU Operation, register with unsigned 6-bit immediate	340
Figure 7-18 Extension ALU Operation, register with signed 12-bit immediate	340
Figure 7-19 Extension ALU Operation, conditional register	340
Figure 7-20 Extension ALU Operation, cc register with unsigned 6-bit immediate.	341
Figure 8-1 Compact Move Format	345
Figure 8-2 Compact Load Format.	345
Figure 8-3 Compact Load and Subtract Format.	346
Figure 8-4 Compact Add Format	346
Figure 8-5 Compact Load and Store Format.	347
Figure 8-6 Compact Load Indexed Format.	347
Figure 8-7 Indexed Jump and Execute Format.	348
Figure 8-8 Load and Add Register-Register Format	349
Figure 8-9 Add/Sub/Shift Register-Immediate Format.	350
Figure 8-10 Dual Register with a 3-bit b Register Format.	351
Figure 8-11 Dual Register with a 3-bit Biased Signed Integer Format	351
Figure 8-12 Long Immediate Operand Format.	352
Figure 8-13 Compact General Operations Register-Register Format	353
Figure 8-14 Compact Single Operand, Jumps, Special Formats	354
Figure 8-15 Compact Zero Operand Instructions	355
Figure 8-16 Compact Load/Store with Offset Format	357

Figure 8-17 Compact Shift/Sub Bit Immediate Format	358
Figure 8-18 Compact Load/Add GP-Relative Format	360
Figure 8-19 Compact Load PCL Relative Format	361
Figure 8-20 Compact Move Immediate Format	362
Figure 8-21 Compact ADD/CMP Immediate Format	363
Figure 8-22 Compact Branch on Compare Register with Zero Format	364
Figure 8-23 Compact Branch Conditionally Format	365
Figure 8-24 Branch Conditionally with cc Field Format	365
Figure 8-25 Compact Branch and Link Unconditionally Format	367
Figure 9-1 Loop Detection and Update Mechanism	618
Figure 9-2 SLEEP Operand	761
Figure 10-1 Example Host Memory Maps, Contiguous Host Memory	849
Figure 10-2 Example Host Memory Maps, Host Memory and Host I/O	849
Figure 11-1 Example 1: MPU_RDB0 Register	858
Figure 11-2 Example 1: MPU_RDP0 Register	858
Figure 11-3 Example 1: MPU_RDB1 Register	858
Figure 11-4 Example 1: MPU_RDP1 Register	858
Figure 11-5 Example 1: MPU_RDB2 Register	858
Figure 11-6 Example 1: MPU_RDP2 Register	858
Figure 11-7 Overlapping Regions Memory Map	859
Figure 11-8 Example 2: MPU_RDB0 Register	859
Figure 11-9 Example 2: MPU_RDP0 Register	859
Figure 11-10 Example 2: MPU_RDB7 Register	859
Figure 11-11 Example 2: MPU_RDP7 Register	859
Figure 11-12 MPU_EN Register	867
Figure 11-13 MPU_ECR Register	870
Figure 11-14 MPU Region Descriptor Base Registers	873
Figure 11-15 MPU Region Descriptor Permissions Registers	874
Figure 11-16 MPU_BUILD Register	878

Figure 12-1 Stack regions	884
Figure 12-2 Stack Checking Scenarios	885
Figure 12-3 USTACK_TOP when ADDR_SIZE == 32.	887
Figure 12-4 USTACK_TOP when ADDR_SIZE < 32.	887
Figure 12-5 USTACK_BASE when ADDR_SIZE == 32.	888
Figure 12-6 USTACK_BASE when ADDR_SIZE < 32.	888
Figure 12-7 KSTACK_TOP when ADDR_SIZE == 32.	889
Figure 12-8 KSTACK_TOP, when ADDR_SIZE < 32	889
Figure 12-9 KSTACK_BASE when ADDR_SIZE == 32.	890
Figure 12-10 KSTACK_BASE when ADDR_SIZE < 32.	890
Figure 12-11 STACK_REGION_BUILD Register	891
Figure 13-1 MEMSEG Register	897
Figure 13-2 Page Descriptor Format	898
Figure 13-3 TLB Page Descriptor Register, TLBPD1	901
Figure 13-4 TLBPD1_HI Register	904
Figure 13-5 TLBIndex Register	905
Figure 13-6 TLBCommand Register	907
Figure 13-7 Process Identity Register, PID Register Description.	915
Figure 13-8 Shared Library Address Space Identifier Register, SASID0, SASID1	917
Figure 13-9 SCRATCH_DATA0 Register	919
Figure 13-10 MMU_BUILD Register	920
Figure 14-1 AUX_ICCM, base address for ICCM (ADDR_SIZE == 32)	924
Figure 14-2 AUX_ICCM, base address for ICCM (ADDR_SIZE == 16)	924
Figure 14-3 AUX_ICCM, base address for ICCM (ADDR_SIZE == m)	924
Figure 14-4 ICCM_BUILD Register	925
Figure 15-1 IC_IVIC Register	929
Figure 15-2 IC_CTRL Register	930
Figure 15-3 IC_LIL Register.	932
Figure 15-4 IC_IVIR Register.	934

Figure 15-5 IC_ENDR Register	936
Figure 15-6 IC_IVIL Register	937
Figure 15-7 IC_RAM_ADDR Register for Direct-Cache-Access (AT==0)	938
Figure 15-8 IC_TAG Register for Reads	941
Figure 15-9 IC_TAG Register for Writes	941
Figure 15-10 IC_DATA Register	945
Figure 15-11 IC_PTAG Register	948
Figure 15-12 IC_PTAG_HI Register	949
Figure 15-13 IC_WAYLOCK Register	950
Figure 15-14 I_CACHE_BUILD Register	951
Figure 16-1 AUX_DCCM, base address for DCCM (ADDR_SIZE == 32)	954
Figure 16-2 AUX_DCCM, base address for DCCM (ADDR_SIZE == 16)	954
Figure 16-3 AUX_DCCM, base address for DCCM (ADDR_SIZE == m)	954
Figure 16-4 DCCM_BUILD Register	956
Figure 16-5 DCCM_BUILD Register	957
Figure 17-1 DC_IVDC Register	963
Figure 17-2 DC_CTRL Register	964
Figure 17-3 DC_LDL Register	966
Figure 17-4 DC_IVDL Register	967
Figure 17-5 DC_FLSH Register	968
Figure 17-6 DC_FLDL Register	969
Figure 17-7 DC_STARTR Register	970
Figure 17-8 DC_ENDR Register	971
Figure 17-9 DC_RAM_ADDR for Cache Controlled RAM Access (AT==1)	972
Figure 17-10 DC_RAM_ADDR Register for Direct-Cache-Access (AT==0)	972
Figure 17-11 DC_TAG for Reads	975
Figure 17-12 DC_TAG for Writes	975
Figure 17-13 DC_DATA	977
Figure 17-14 AUX_VOLATILE Register	978

Figure 17-15 Volatile Space	980
Figure 17-16 DC_PTAG_HI	981
Figure 17-17 D_CACHE_BUILD	982
Figure 18-1 CSM_ENABLE Register	986
Figure 18-2 CSM_ECC_CTRL Register	988
Figure 18-3 CSM_SBE_CNT Register	990
Figure 18-4 CSM_ECC_INI Register	991
Figure 18-5 CSM_BASE_SYS_ADDR_L Register	994
Figure 18-6 CSM_BASE_SYS_ADDR_H Register	995
Figure 18-7 CSM_BUILD Register	996
Figure 19-1 SLC_AUX_CACHE_CONFIG REGISTER	1001
Figure 19-2 SLC_ECC_CONFIG Register	1003
Figure 19-3 SLC_AUX_CACHE_CTRL Register	1005
Figure 19-4 SLC_AUX_CACHE_FLUSH Register	1010
Figure 19-5 SLC_AUX_CACHE_INV Register	1011
Figure 19-6 Cache Way-Lock Register	1012
Figure 19-7 SLC_AUX_LINE_LOCK Register	1014
Figure 19-8 SLC_AUX_LINE_LOCK Register	1015
Figure 19-9 SLC_AUX_LINE_INV Register	1016
Figure 19-10 SLC_AUX_LINE_INV1 Register	1017
Figure 19-11 SLC_AUX_LINE_FLUSH Register	1018
Figure 19-12 SLC_AUX_LINE_FLUSH1 Register	1019
Figure 19-13 SLC_AUX_RGN_START Register	1020
Figure 19-14 SLC_AUX_RGN_START1 Register	1021
Figure 19-15 SLC_AUX_RGN_END Register	1022
Figure 19-16 SLC_AUX_RGN_END1 Register	1023
Figure 19-17 SLC_AUX_LINE_ADDR Register	1024
Figure 19-18 SLC_AUX_LINE_ADDR1 Register	1025
Figure 19-19 SLC_AUX_DIRECT_IDX Register	1026

Figure 19-20 SLC_AUX_TAG_DATA Register	1028
Figure 19-21 SLC_AUX_TAG_DATA1 Register	1029
Figure 19-22 SLC_AUX_STATUS_DATA Register	1030
Figure 19-23 SLC_AUX_DATA0 Register.....	1031
Figure 19-24 SLC_AUX_DATA1 Register.....	1032
Figure 19-25 SLC_AUX_DATA2 Register.....	1033
Figure 19-26 SLC_AUX_DATA3 Register.....	1034
Figure 19-27 SLC_AUX_FAULT_ADDR Register	1035
Figure 19-28 SLC_AUX_FAULT_ADDR1 Register	1036
Figure 19-29 SLC_AUX_FAULT_STAT REGISTER.....	1037
Figure 19-30 SLC_AUX_PM_CMD Register.....	1040
Figure 19-31 SLC_AUX_PM_EVENT Register.....	1042
Figure 19-32 SLC_AUX_PM_OVF Register	1046
Figure 19-33 SLC_AUX_PM_CNT0 Register	1047
Figure 19-34 SLC_AUX_PM_CNT1 Register	1048
Figure 19-35 SLC_AUX_ECC_SBE_CNT Register.....	1049
Figure 19-36 SLC_AUX_ECC_CTRL Register	1051
Figure 19-37 SLC_BUILD Register	1053
Figure 20-1 SM_CONFIG_REG.....	1058
Figure 21-1 ECC_SBE_COUNT Register	1060
Figure 21-2 ECC_SBE_COUNT Register.....	1060
Figure 21-3 ERP_CTRL Register.....	1064
Figure 21-4 ERP_CTRL Register.....	1064
Figure 21-5 ERP_BUILD Register.....	1067
Figure 21-6 ERP_BUILD Register.....	1069
Figure 22-1 DMP_PER_AUX, start of peripheral region.....	1074
Figure 22-2 DMP_PER_AUX, start of peripheral region.....	1074
Figure 22-3 Peripheral Regions Based on Base and Limit Values	1075
Figure 22-4 DMP_PP_BUILD Register	1077

Figure 22-5 DMP_PP_BUILD Register	1077
Figure 23-1 WB_CTRL	1080
Figure 23-2 WB_STATUS	1082
Figure 23-3 WB_BUILD	1083
Figure 24-1 BPU_FLUSH Register	1088
Figure 24-2 BPU_CTRL Register	1089
Figure 24-3 BC_RAM_ADDR Register	1092
Figure 24-4 BC_RAM_INFO Register	1094
Figure 24-5 BC_RAM_BTA Register	1097
Figure 24-6 PT_RAM_ADDR Register	1099
Figure 24-7 PT_RAM_DATA Register when FETCH_BLOCK==128	1101
Figure 24-8 PT_RAM_DATA Register when FETCH_BLOCK==64	1101
Figure 24-9 BPU_BUILD Register	1103
Figure 24-10 BPU RAM Direct Access	1108
Figure 24-11 BPU BC and PT RAM DUMP	1109
Figure 25-1 LPB_CTRL Register	1114
Figure 25-2 LPB_BUILD Register	1115
Figure 26-1 DEBUG Register	1118
Figure 26-2 DEBUGI Register	1122
Figure 27-1 Actionpoint Build Register	1130
Figure 27-2 Actionpoint 0 Match Value Register	1131
Figure 27-3 Actionpoint 0 Match Mask Register	1132
Figure 27-4 Actionpoint 0 Control Register	1133
Figure 27-5 Actionpoint 1 Match Value Register	1138
Figure 27-6 Actionpoint 1 Match Mask Register	1139
Figure 27-7 Actionpoint 1 Control Register	1140
Figure 27-8 Actionpoint 2 Match Value Register	1141
Figure 27-9 Actionpoint 2 Match Mask Register	1142
Figure 27-10 Actionpoint 2 Control Register	1143

Figure 27-11 Actionpoint 3 Match Value Register	1144
Figure 27-12 Actionpoint 3 Match Mask Register	1145
Figure 27-13 Actionpoint 3 Control Register	1146
Figure 27-14 Actionpoint 4 Match Value Register	1147
Figure 27-15 Actionpoint 4 Match Mask Register	1148
Figure 27-16 Actionpoint 4 Control Register	1149
Figure 27-17 Actionpoint 5 Match Value Register	1150
Figure 27-18 Actionpoint 5 Match Mask Register	1151
Figure 27-19 Actionpoint 5 Control Register	1152
Figure 27-20 Actionpoint 6 Match Value Register	1153
Figure 27-21 Actionpoint 6 Match Mask Register	1154
Figure 27-22 Actionpoint 6 Control Register	1155
Figure 27-23 Actionpoint 7 Match Value Register	1156
Figure 27-24 Actionpoint 7 Match Mask Register	1157
Figure 27-25 Actionpoint 7 Control Register	1158
Figure 27-26 Actionpoint Isolation Enable Register	1159
Figure 27-27 AP_WP_PC, Watchpoint Program Counter Register when PC_SIZE = 32.....	1160
Figure 27-28 AP_WP_PC Register when PC_SIZE < 32	1160
Figure 27-29 SmaRT Operation.....	1161
Figure 27-30 SMART_BUILD Value.....	1163
Figure 27-31 SMART_CONTROL Register.....	1164
Figure 27-32 SMART_DATA Register.....	1166
Figure 27-33 SRC_ADDR Value	1166
Figure 27-34 DEST_ADDR Value Register	1167
Figure 27-35 FLAGS Value Register.....	1167
Figure 27-36 DB_STATUS Register	1170
Figure 27-37 DB_CMD Register	1171
Figure 27-38 DB_ADDR Register	1172
Figure 27-39 DB_DATA Register	1173

Figure 27-40 DB_RESET Register	1174
Figure 27-41 Debug Interface Timing Diagram	1175
Figure 28-1 CC_INDEX Register.	1180
Figure 28-2 CC_NAME0 Register.	1181
Figure 28-3 CC_NAME1 Register.	1182
Figure 28-4 PCT_COUNTL Register	1183
Figure 28-5 PCT_COUNTH Register	1184
Figure 28-6 PCT_SNAPL Register	1185
Figure 28-7 PCT_SNAPH Register	1186
Figure 28-8 PCT_CONFIG Register	1187
Figure 28-9 PCT_CONTROL Register	1188
Figure 28-10 PCT_INDEX Register	1190
Figure 28-11 PCT_RANGEL Register	1191
Figure 28-12 PCT_RANGEH Register	1192
Figure 28-13 PCT_UFLAGS Register	1193
Figure 28-14 PCT_INT_CNTRL Register	1195
Figure 28-15 PCT_INT_CNTH Register	1196
Figure 28-16 PCT_INT_CTRL Register	1197
Figure 28-17 PCT_INT_ACT Register	1198
Figure 28-18 PCT_BUILD Register	1199
Figure 28-19 CC_BUILD Register	1200
Figure 29-1 Interrupt Generated after Timer Reaches Limit Value.	1202
Figure 29-2 Timer 0 Count Value Register	1204
Figure 29-3 Timer 0 Control Register.	1205
Figure 29-4 Timer 0 Limit Value Register	1207
Figure 29-5 Timer 1 Count Value Register	1208
Figure 29-6 Timer 0 Control Register.	1209
Figure 29-7 Timer 1 Limit Value Register	1210
Figure 29-8 AUX_RTC_CTRL Register	1211

Figure 29-9 State Machine of A1 and A0 bits in RTC Control Register	1212
Figure 29-10 RTC Count Low Register	1213
Figure 29-11 AUX_RTC_HIGH Register	1214
Figure 29-12 TIMER_BUILD Register	1215
Figure 30-1 CLUSTER_ID Register	1219
Figure 30-2 GLOBAL_CLK_GATE_DIS Register	1220
Figure 30-3 GLOBAL_CLK_GATE_DIS Register	1220
Figure 30-4 CLUSTER_VCCM_QoS Register	1222
Figure 30-5 CC_NMI_STATUS Register	1224
Figure 30-6 CC_NMI_CAUSE Register	1226
Figure 30-7 CLUSTER_BUILD Register	1230
Figure 30-8 CLUSTER_BUILD Register	1230
Figure 31-1 BIU_IO_0_COH_ENABLE Register	1234
Figure 31-2 BIU_IO_0_COH_PARTIAL Register	1235
Figure 31-3 BIU_IO_0_COH_AP0_BASE Register when Physical Address Extension is Disabled	1236
Figure 31-4 BIU_IO_0_COH_AP0_BASE Register when Physical Address Extension is Enabled	1236
Figure 31-5 BIU_IO_0_COH_AP0_SIZE Register when Physical Address Extension is Disabled	1237
Figure 31-6 BIU_IO_0_COH_AP0_SIZE Register when Physical Address Extension is Enabled	1237
Figure 31-7 BIU_BUILD Register	1238
Figure 31-8 BIU_BUILD Register	1238
Figure 32-1 PDM_PSTAT Register	1243
Figure 32-2 PDM_PSTAT Register	1243
Figure 32-3 RTT_PDM_PSTAT Register	1245
Figure 32-4 PDM_PMODE Register	1246
Figure 32-5 PDM_SCU_DIS_IND Register	1247
Figure 32-6 PDM_CC_PMODE Register	1248
Figure 32-7 PDM_CC_PSTAT Register for a Four-Core Cluster	1249
Figure 32-8 PDM_CC_PSTAT Register for a Four-Core Cluster	1249
Figure 32-9 PDM_CSM_PMODE Register	1252

Figure 32-10 PDM_CSM_PSTAT Register	1253
Figure 32-11 PDM_CSM_PSTAT Register	1253
Figure 32-12 PDM_SLC_PMODE Register	1255
Figure 32-13 PDM_SLC_PSTAT Register	1256
Figure 32-14 PDM_CSM_P_PMODE Register	1257
Figure 32-15 PDM_CSM_P_PSTAT Register	1259
Figure 32-16 PDM_SLC_P_PMODE Register	1262
Figure 32-17 PDM_SLC_P_PSTAT Register	1264
Figure 32-18 PDM_CSM_SM_P_CFG Register	1267
Figure 32-19 DVFS_CLUSTER_PL Register	1269
Figure 32-20 EXEC_CTRL Register	1270
Figure 32-21 PDM_DVFS_BUILD Register	1271
Figure 33-1 CONNECT_CMD Register	1275
Figure 33-2 CONNECT_WDATA Register	1276
Figure 33-3 CONNECT_READBACK Register	1277
Figure 33-4 CONNECT_SYSTEM_BUILD Register	1278
Figure 33-5 CONNECT_SEMA_BUILD Register	1280
Figure 33-6 CONNECT_MESSAGE_BUILD Register	1281
Figure 33-7 CONNECT_PMU_BUILD Register	1283
Figure 33-8 CONNECT_IDU_BUILD Register	1284
Figure 33-9 CONNECT_GFRC_BUILD Register	1285
Figure 33-10 CONNECT_IVC_BUILD Register	1286
Figure 33-11 CONNECT_ICI_BUILD Register	1287
Figure 33-12 CONNECT_ICD_BUILD Register	1288
Figure 33-13 CONNECT_ASI_BUILD Register	1289
Figure 33-14 CONNECT_PDM_BUILD Register	1290
Figure 34-1 Cluster DMA Client – Server Architecture	1293
Figure 34-2 Cluster DMA High-Level Architecture	1295
Figure 34-3 DMA_C_CTRL_AUX Register	1298

Figure 34-4 DMA_C_CHAN_AUX Register	1299
Figure 34-5 DMA_C_SRC_AUX Register	1300
Figure 34-6 DMA_C_SRC_HI_AUX Register	1301
Figure 34-7 DMA_C_DST_AUX Register	1302
Figure 34-8 DMA_C_DST_HI_AUX Register	1303
Figure 34-9 DMA_C_ATTR_AUX Register	1304
Figure 34-10 DMA_C_LEN_AUX Register	1306
Figure 34-11 DMA_C_HANDLE_AUX Register	1307
Figure 34-12 DMA_C_EVSTAT_AUX Register	1308
Figure 34-13 DMA_C_EVSTAT_CLR_AUX Register	1309
Figure 34-14 DMA_C_STATUS_AUX Register	1310
Figure 34-15 DMA_C_INTSTAT_AUX Register	1311
Figure 34-16 DMA_C_INTSTAT_CLR_AUX Register	1313
Figure 34-17 DMA_C_ERRHANDLE_AUX Register	1314
Figure 34-18 DMA_S_CTRL_AUX Register	1315
Figure 34-19 DMA_S_DONESTATD_AUX Register	1316
Figure 34-20 DMA_S_DONESTATD_AUX Register	1317
Figure 34-21 DMA_S_TAILC_AUX Register	1318
Figure 34-22 DMA_S_MIDC_AUX Register	1319
Figure 34-23 DMA_S_HEADC_AUX Register	1320
Figure 34-24 DMA_S_BASEC_AUX Register	1321
Figure 34-25 DMA_S_LASTC_AUX Register	1322
Figure 34-26 DMA_S_PRIORC_AUX Register	1323
Figure 34-27 DMA_S_STATC_AUX Register	1324
Figure 34-28 DMA_BUILD Register	1327
Figure 35-1 Core Register Map Summary	1329
Figure 35-2 ACCH for Double-Precision FPU operations	1330
Figure 35-3 ACCL for Double-Precision FPU operations	1330
Figure 35-4 ACCL for Single-Precision FPU Operations	1330

Figure 35-5 FPU_CTRL Register	1332
Figure 35-6 FPU_STATUS Register	1334
Figure 35-7 FPU_BUILD Register	1338
Figure 37-1 RPU Functionality	1343
Figure 37-2 RPU Mapping	1344
Figure 37-3 RPU_CMND Register	1348
Figure 37-4 RPU_ADDR Register	1349
Figure 37-5 RPU_DATA Register	1351
Figure 37-6 RPU_ENABLE Register	1352
Figure 37-7 RPU_MODE Register	1353
Figure 37-8 RPU_STATUS Register	1354
Figure 37-9 RPU_ROM_BASE Register	1355
Figure 37-10 RPU_PATCH_TABLE_BASE Register	1356
Figure 37-11 PATCH_OFFSET _n Register	1357
Figure 37-12 RPU_BUILD Register	1358
Figure 38-1 FastMath Extension Pack auxiliary Register Map	1363
Figure 38-2 FMP_BUILD Register	1364
Figure 39-1 Vector Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==0.	1427
Figure 39-2 Vector Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==1.	1427
Figure 39-3 Vector Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==0.	1427
Figure 39-4 Vector Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==1.	1427
Figure 39-5 Wide Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==0.	1427
Figure 39-6 Wide Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==1.	1428
Figure 39-7 Wide Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==0.	1428
Figure 39-8 Wide Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==1.	1428
Figure 40-1 ACC0_LO Register	1432
Figure 40-2 ACC0_GLO Register	1433
Figure 40-3 ACC0_HI Register	1434
Figure 40-4 ACC0_GHI Register	1435

Figure 40-5 DSP_BFLY0 Register in Little-Endian Mode	1436
Figure 40-6 DSP_BFLY0 Register in Big-Endian Mode	1436
Figure 40-7 DSP_FFT_CTRL Register	1437
Figure 40-8 DSP_CTRL Register	1438
Figure 40-9 DSP_BUILD Register	1440
Figure 41-1 AGU_AUX_APx Register	1445
Figure 41-2 AGU_AUX_OSx Register	1448
Figure 41-3 AGU_AUX_MODx Register with Offset Register Increment	1449
Figure 41-4 AGU_AUX_MODx Register with Immediate Offset	1449
Figure 41-5 AGU_AUX_MODx Register with Bit Reverse Increment Using Offset Register	1449
Figure 41-6 AGU_AUX_MODx Register with Bit Reverse Increment Using Immediate Offset	1449
Figure 41-7 AGU_AUX_MODx Register with Modulo Wrapping Using Offset and Wrap Registers	1449
Figure 41-8 AGU_AUX_MODx Register with Modulo Wrapping Using Immediate Offset and Wrap	1450
Figure 41-9 AGU_AUX_MODx with Modulo Wrapping Using Offset Register and Immediate Wrap	1450
Figure 41-10 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset Register	1450
Figure 41-11 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset Immediate ...	1450
Figure 41-12 AGU_BUILD Register	1452
Figure C-1 CONNECT_READBACK Register Format for CMD_CHECK_CORE_ID Command	1952
Figure C-2 CONNECT_READBACK Register Format for CMD_INTRPT_READ_STATUS Command	1954
Figure C-3 CONNECT_READBACK Register Format for CMD_INTRPT_CHECK_SOURCE Command ...	1955
Figure C-4 CONNECT_WDATA Register Format for CMD_INTRPT_SET_PULSE_CNT Command	1958
Figure C-5 CONNECT_READBACK Register Format for CMD_INTRPT_READ_PULSE_CNT Command .	1959
Figure C-6 CONNECT_READBACK Register Format for CMD_SEMA_CLAIM_AND_READ Command .	1959
Figure C-7 CONNECT_WDATA Register Format for CMD_MSG_SRAM_SET_ADDR Command	1961
Figure C-8 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_ADDR Command ...	1962
Figure C-9 CONNECT_WDATA Register Format for Set-Offset Command	1962
Figure C-10 CONNECT_READBACK Register Format for Read-Offset Command	1963
Figure C-11 CONNECT_WDATA Register Format for CMD_MSG_SRAM_WRITE Command	1964
Figure C-12 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ Command	1966

Figure C-13 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_INC Command	1967
Figure C-14 CONNECT_WDATA Register Format for CMD_MSG_SET_ECC_CTRL Command	1968
Figure C-15 CONNECT_READBACK Register Format for CMD_MSG_READ_ECC_SBE_CNT Command.	1970
Figure C-16 CONNECT_WDATA Register Format for CMD_DEBUG_RESET Command	1971
Figure C-17 CONNECT_READBACK Register Format for CMD_DEBUG_READ_MASK Command	1974
Figure C-18 CONNECT_READBACK Register Format for CMD_DEBUG_READ_SELECT Command. . . .	1976
Figure C-19 CONNECT_READBACK Register Format for CMD_DEBUG_READ_EN	1976
Figure C-20 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CMD Command	1977
Figure C-21 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CORE Command	1978
Figure C-22 CONNECT_READBACK Register Format for CMD_GFRC_READ_LO Command.	1979
Figure C-23 CONNECT_READBACK Register Format for CMD_GFRC_READ_HI Command	1980
Figure C-24 CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command	1981
Figure C-25 CONNECT_WDATA Register Format for Command CMD_GFRC_SET_CORE.	1982
Figure C-26 CONNECT_READBACK Register Format for Command CMD_GFRC_READ_CORE	1983
Figure C-27 CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command	1984
Figure C-28 CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command.	1985
Figure C-29 CONNECT_WDAT Register Format for CMD_PDM_SET_PM Command	1986
Figure C-30 CONNECT_READBACK Register Format for CMD_PDM_READ_STATUS Command.	1987
Figure C-31 CONNECT_WDATA Register Format for CMD_IVC_SET Command	1988
Figure C-32 CONNECT_READBACK Register Format for CMD_IVC_READ Command	1989
Figure C-33 CONNECT_READBACK Register Format for CMD_PMU_SET_PUCNT Command	1989
Figure C-34 CONNECT_READBACK Register Format for CMD_PMU_SET_RSTCNT Command.	1991
Figure C-35 CONNECT_READBACK Register Format for CMD_PMU_SET_PDCNT Command	1992
Figure C-36 CONNECT_READBACK Register Format for CMD_IDU_READ_ENABLECommand.	1994
Figure C-37 CONNECT_WDATA Register Format for CMD_IDU_SET_MODE Command.	1995
Figure C-38 CONNECT_WDATA Register Format for CMD_IDU_SET_DEST Command.	1996
Figure C-39 CONNECT_READBACK Register Format for CMD_IDU_CHECK_STATUS Command	1999
Figure C-40 CONNECT_READBACK Register Format for CMD_IDU_CHECK_SOURCE command	2001
Figure C-41 CONNECT_WDATA Register Format for CMD_IDU_SET_MASK command	2003

Figure C-42 CONNECT_READBACK Register Format for CMD_IDU_CHECK_FIRST Command 2004

List of Examples

Example 3-1 Null Instruction Format	122
Example 4-1 Correct LP_COUNT Setup through a Register	132
Example 5-1 Setting Interrupt Priority or Enable (STATUS32.IE) of an Interrupt	214
Example 5-2 Regular Interrupt Entry.....	220
Example 5-3 Regular Interrupt Exit	224
Example 5-4 Pass Control to an Handler or Perform a Soft Reset.....	229
Example 5-5 Exception Vector Code	229
Example 5-6 Exception in a Delay Slot	253
Example 9-1 To obtain a semaphore using EX.....	497
Example 9-2 To Release Semaphore using ST	498
Example 9-3 Example Loop Code	617
Example 9-4 Use of Zero Overhead Loop.....	620
Example 9-5 Sleep placement in code	762
Example 9-6 Enable Interrupts and Sleep.....	763
Example 9-7 WEVT placement in code	841
Example 9-8 WLFC Placement in Code.....	843
Example 28-1 Example Operations	1193
Example 41-1 Pseudocode for Updating an Address Pointer Value.....	1441
Example 41-2 Assembly Code Using Address Generators.....	1443
Example C-1 Segment Program of Scenario 1 – Core1	2005
Example C-2 Segment Program of Scenario 1 – Core2	2005
Example C-3 Segment Program of Scenario 2 - Core0, Core1, Core2.....	2006
Example C-4 Segment Program of Scenario 2 - Core3	2006

Example C-5 Segment Program of Scenario 1- Core1	2008
Example C-6 Segment Program of Scenario 2 - Core 2	2008
Example C-7 Segment Program of Scenario 1 - Core1	2009
Example C-8 Conditionally Halt Cores Scenario	2010
Example C-9 Selectively Halt Cores Scenario	2010
Example C-10 Selectively Run Cores Scenario	2010
Example C-11 Selectively Reset Cores Scenario	2011
Example C-12 Segment Program of Scenario 1 - Core2	2011
Example C-13 Segment Program to Read Value of GFRC	2012
Example C-14 Segment Program of Common Interrupt 1 Generated by Core 0	2013

List of Tables

Table 1-1 Differences Between ARC HS4x and ARC HS3x Processors	70
Table 1-1 Auxiliary Register List	75
Table 1-2 Build Configuration Registers	84
Table 2-1 Exception Vectors and Cause Codes.....	87
Table 2-1 ARCV2 Programming Model Configuration Options	99
Table 2-2 ARCV2 Instruction Set Options	102
Table 3-1 Half-Precision Floating-Point Data Field Descriptions	114
Table 3-2 Single-Precision Floating-Point Data Field Descriptions.....	114
Table 3-3 Double-Precision Floating-Point Data Field Descriptions.....	115
Table 3-4 Instruction Variants.....	121
Table 3-5 Operand Description.....	122
Table 4-1 Core Register Set	126
Table 4-2 16-bit Instruction Register Encoding.....	127
Table 4-3 Current ABI Register Usage.....	128
Table 4-4 Multiple Bank Configuration.....	129
Table 4-5 Baseline Auxiliary Register Set	134
Table 4-6 Key to Auxiliary Register Access Permissions for LR and SR Instructions	136
Table 4-7 Zero Overhead Loop Auxiliary Registers (lpc_size > 0)	136
Table 4-8 Indexed Table Auxiliary Register	137
Table 4-9 XPU, User Extension Permission Auxiliary Register	137
Table 4-10 Extension Flags Register, XFLAGS	137
Table 4-11 LP_START Field Description.....	139
Table 4-12 LP_END Field Description.....	140

Table 4-13 IDENTITY Field Descriptions	141
Table 4-14 ARCV2 Addressable Range vs. PC Size	143
Table 4-15 PC Field Description	144
Table 4-16 STATUS32 Bit-Field Definitions, Read / Write Accessibility , and Reset Value	146
Table 4-17 AUX_USER_SP Field Description	152
Table 4-18 AUX_IRQ_CTRL Field Description	153
Table 4-19 INT_VECTOR_BASE Field Description	154
Table 4-20 AUX_IRQ_ACT Field Description	155
Table 4-21 IRQ_PRIORITY_PENDING Field Description	156
Table 4-22 IRQ_PRIORITY Field Description	158
Table 4-23 DATA_MEM_ATTR Field Descriptions	160
Table 4-24 ECR Field Description	169
Table 4-25 EFA Field Description	171
Table 4-26 ICAUSE Field Description	172
Table 4-27 IRQ_SELECT Field Description	173
Table 4-28 IRQ_ENABLE Field Description	174
Table 4-29 IRQ_TRIGGER Field Description	175
Table 4-30 IRQ_STATUS Field Description	176
Table 4-31 IRQ_PULSE_CANCEL Field Description	181
Table 4-32 IRQ_PENDING Field Description	182
Table 4-33 Build Configuration Registers	184
Table 4-34 BCR_VER Field Descriptions	185
Table 4-35 BTA_LINK_BUILD Field Descriptions	186
Table 4-36 VECBASE_AC_BUILD Field Descriptions	187
Table 4-37 Reset Value of the Interrupt Vector Base	187
Table 4-38 RF_BUILD Field Descriptions	189
Table 4-39 MULTIPLY_BUILD Field Descriptions	191
Table 4-40 SWAP_BUILD Field Descriptions	192
Table 4-41 NORM_BUILD Field Descriptions	193

Table 4-42 MINMAX_BUILD field descriptions	194
Table 4-43 BARREL_BUILD Field Descriptions.....	195
Table 4-44 ISA_CONFIG Field Descriptions.....	196
Table 4-45 IRQ_BUILD Field Descriptions	199
Table 4-46 ISA_PROFILE field descriptions	201
Table 4-47 MICRO_ARCH_BUILD Field Descriptions.....	203
Table 5-1 Overview of Privileges in Kernel and User Modes	206
Table 5-2 Interrupt Configuration Options.....	209
Table 5-3 Interrupt/Exception Vectors	228
Table 5-4 Exception Vectors and Cause Codes.....	233
Table 5-5 EFA and ERET Entries for Exception Types	238
Table 5-6 Exception and Interrupt Exit Modes.....	251
Table 6-1 Top-level Formats of the ARCV2 Instruction Set	256
Table 6-2 Assignment of top-level instruction formats to major opcodes in each instruction set profile	258
Table 6-3 Summary of 32-bit instruction formats.....	260
Table 6-4 Summary of 16-bit instruction formats.....	261
Table 6-5 Key for 32-bit Addressing Modes and Encoding Conventions	262
Table 6-6 Key for 16-bit Addressing Modes and Encoding Conventions	263
Table 6-7 Condition Codes	264
Table 6-8 Condition Codes	265
Table 6-9 Delay Slot Modes.....	266
Table 6-10 Encoding Static Branch Predictions for BRcc, BBIT0, and BBIT1	266
Table 6-11 Address Writeback Modes	267
Table 6-12 Scaling Shift	268
Table 6-13 Cache Bypass Modes.....	268
Table 6-14 Load Store Data Sizes	268
Table 6-15 Load Data Extend Mode.....	269
Table 6-16 Store Data Source Mode	269
Table 6-17 Instruction Syntax Convention	270

Table 6-18 Integer addition, subtraction and comparison operations	273
Table 6-19 Move operations	274
Table 6-20 Bit-wise logical operations	275
Table 6-21 Bit-mask operations.....	275
Table 6-22 Single Bit Shift and Rotate Operations	276
Table 6-23 Multi-bit Shift Operations.....	277
Table 6-24 Shift-assist operations	277
Table 6-25 Selection operations.....	278
Table 6-26 Byte-swapping operations	278
Table 6-27 Bit-scanning operations	278
Table 6-28 Relational Comparison Operations.....	278
Table 6-29 Integer Multiply, MAC and Divide Operations	279
Table 6-30 Dual and Quad Integer Multiply and MAC Operations	280
Table 6-31 Integer Vector ADD, SUB, MPY operations	281
Table 6-32 Memory Access Instructions	283
Table 6-33 PREFETCH Instruction Encodings	288
Table 6-34 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions.....	288
Table 6-35 Auxiliary Register Operations.....	289
Table 6-36 Control Flow Operations	289
Table 6-37 Delay Slot Execution Modes.....	291
Table 6-38 Branch on Compare/Test Mnemonics	293
Table 6-39 Branch on Compare Pseudo Mnemonics, Register-Register.....	293
Table 6-40 Branch on Compare Pseudo Mnemonics, Register-Immediate	293
Table 6-41 Delay Slot Execution Modes.....	294
Table 6-42 Special Instructions	295
Table 6-43 Floating-Point Unit Instructions, 0x06	299
Table 6-44 Instruction Mnemonics	302
Table 6-45 Basic Saturating Arithmetic Operations.....	303
Table 6-46 Vector Unpacking Operations	304

Table 6-47 Vector ALU Operations	306
Table 6-48 Accumulator Operations	307
Table 6-49 Vector 16x16 MAC Operations	307
Table 6-50 Dual 16x16 MAC Operations.....	309
Table 6-51 32x32 and 16x16 MAC Operations	309
Table 6-52 Dual 16x8 MAC Operations.....	311
Table 6-53 32x16 MAC Operations.....	311
Table 6-54 Complex Operations	314
Table 6-55 Vector 64-Bit Operations.....	315
Table 6-56 Divide and Square-Root Operations.....	316
Table 6-57 Address Generation Instructions.....	317
Table 7-1 Branch on Compare/Bit Test Register-Register	322
Table 7-2 Hierarchical Sub-formats of the General Operations Formats	327
Table 7-3 Operand Sub-format Indicators.....	328
Table 7-4 Opcode assignment for DOP instructions in major op-code 0x04 (F32_GEN4).....	331
Table 7-5 Special DOP sub-formats in F32_GEN4.....	332
Table 7-6 Opcode assignment for SOP instructions in major op-code 0x04 (F32_GEN4)	334
Table 7-7 Opcode assignment for ZOP instructions in major op-code 0x04 (F32_GEN4).....	334
Table 7-8 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5)	335
Table 7-9 Opcode assignment for SOP instructions in major op-code 0x05 (F32_EXT5).....	336
Table 7-10 Opcode assignment for ZOP instructions in major op-code 0x05 (F32_EXT5).....	337
Table 7-11 Opcode assignment for DOP instructions in major op-code 0x06 (F32_EXT6)	338
Table 7-12 Opcode assignment for SOP instructions in major op-code 0x06 (F32_EXT6).....	338
Table 7-13 Opcode assignment for ZOP instructions in major op-code 0x06 (F32_EXT6).....	339
Table 7-14 List of FastMath extension pack instructions	341
Table 8-1 16-Bit Indexed Jump and Link Operations	348
Table 8-2 16-Bit, LD / ADD Register-Register	349
Table 8-3 16-Bit, ADD/SUB Register-Immediate.....	350
Table 8-4 16-Bit MOV/CMP/ADD with High Register	352

Table 8-5 DOP_format 16-Bit General Operations	353
Table 8-6 16-Bit Single Operand Instructions	355
Table 8-7 16-Bit Zero Operand Instructions	356
Table 8-8 Summary of 16-Bit Load and Store Instructions with Offset	357
Table 8-9 16-Bit Shift/SUB/Bit Immediate	358
Table 8-10 Opcodes for Stack-based Operations	359
Table 8-11 16-Bit GP Relative Instructions	360
Table 8-12 16-Bit ADD/CMP Immediate	363
Table 8-13 16-Bit Branch on Compare	364
Table 8-14 16-Bit Branch, Branch Conditionally	365
Table 8-15 16-Bit Branch Conditionally	366
Table 9-1 List of Instructions	370
Table 9-2 Delay Slot Modes for the Jump and Link Instructions	592
Table 9-3 When Reads and Writes Take Effect	619
Table 9-4 PREFETCH Instruction Encodings	696
Table 9-5 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions	697
Table 9-6 SLEEP Operand	761
Table 10-1 Host Accesses to the ARCV2-based processor	850
Table 10-2 Single Step Flags in Debug Register	851
Table 11-1 Example 1 -- Contiguous Regions Memory Map	857
Table 11-2 Priority Configurations	860
Table 11-3 Setting ECR Register based on EV_ProtV Exception	862
Table 11-4 ECR parameter values for EV_ProtV sources	863
Table 11-5 MPU Register Set	864
Table 11-6 MPU Enable Register	867
Table 11-7 MPU Exception Cause Register	871
Table 11-8 MPU Region Descriptor Base Registers Field Description	873
Table 11-9 MPU Region Descriptor Permission Registers Field Description	875
Table 11-10 Build Configuration Registers	877

Table 11-11 MPU_BUILD Field Description	878
Table 12-1 Stack Checking Specification	885
Table 12-2 Stack Region Checking Auxiliary Registers	886
Table 12-3 STACK_REGION_BUILD Field Descriptions	891
Table 13-1 Special Purpose Registers for TLB Control	895
Table 13-2 TLBIndex Addresses	905
Table 13-3 Result Code, Read Only	906
Table 13-4 TLB Command Register Command List	907
Table 13-5 MMU_BUILD Register Fields	920
Table 14-1 ICCM Registers	923
Table 14-2 ICCM_BUILD Field Descriptions	925
Table 15-1 Instruction Cache Auxiliary Registers	927
Table 15-2 IC_CTRL Control Flags	930
Table 15-3 IC_RAM_ADDR for Cache Controlled RAM Access (AT==1)	938
Table 15-4 IC_RAM_ADDR for Aliased Configurations	938
Table 15-5 IC_RAM_ADDRESS Register Behavior	939
Table 15-6 IC_TAG Register for Reads in an Aliased Configuration	941
Table 15-7 IC_TAG Register Behavior	942
Table 15-8 IC_DATA Register Behavior	946
Table 15-9 I_CACHE_BUILD Field Descriptions	951
Table 16-1 DCCM Auxiliary Registers	953
Table 16-2 DCCM_BUILD Field Descriptions	956
Table 16-3 DCCM_BUILD Field Descriptions	957
Table 17-1 Data Cache Auxiliary Registers	961
Table 17-2 DC_IVDC Field Descriptions	963
Table 17-3 DC_CTRL Control Flags	964
Table 17-4 DC_FLSH Control Flags	968
Table 17-5 DC_RAM_ADDRESS Register Behavior	973
Table 17-6 DC_TAG Control Flags and Update Conditions	975

Table 17-7 AUX_VOLATILE Field Description	980
Table 17-8 Build Configuration Registers	981
Table 17-9 D_CACHE_BUILD Field Descriptions	982
Table 18-1 CSM Registers	985
Table 18-2 CSM_ENABLE Bit Field Description	987
Table 18-3 CSM_ECC_CTRL Bit Field Description	988
Table 18-4 CSM_SBE_CNT Bit Field Description	990
Table 18-5 CSM_ECC_INI Bit Field Description	991
Table 18-6 CSM_BUILD Field Descriptions	996
Table 19-1 L2 Auxiliary Registers	999
Table 19-2 SLC_AUX_CACHE_CONFIG Field Description	1001
Table 19-3 SLC_ECC_CONFIG Field Description	1003
Table 19-4 Data Array ECC Configuration	1003
Table 19-5 SLC_AUX_CACHE_CTRL Register Field Description	1005
Table 19-6 SLC_AUX_CACHE_FLUSH Register Field Description	1010
Table 19-7 SLC_AUX_CACHE_INV Register Field Description	1011
Table 19-8 SLC_AUX_DIRECT_IDX Field Description	1026
Table 19-9 SLC_AUX_TAG_DATA Register Field Description	1028
Table 19-10 SLC_AUX_STATUS_DATA Register Field Description	1030
Table 19-11 SLC_AUX_FAULT_STAT Register Field Description	1038
Table 19-12 SLC_AUX_PM_CMD Field Description	1040
Table 19-13 SLC_AUX_PM_EVENT Field Description	1042
Table 19-14 Performance Counter Event Listing	1042
Table 19-15 SLC_AUX_ECC_SBE_CNT Bit Field Description	1049
Table 19-16 SLC_AUX_ECC_CTRL Bit Field Description	1051
Table 19-17 L2_BUILD Field Description	1053
Table 20-1 Shared Memory Configurations	1056
Table 20-2 Shared Memory Configuration Register Field Description	1058
Table 21-1 Error Protection Auxiliary Registers	1059

Table 21-2 ECC_SBE_COUNT Bit Field Description	1060
Table 21-3 ERP_CTRL Bit Field Description	1064
Table 21-4 ERP_BUILD Register	1067
Table 21-5 ERP_BUILD Register	1070
Table 22-1 DMP Registers	1073
Table 22-2 DMP_PER_AUX Field Descriptions	1076
Table 22-3 DMP_PP_BUILD Field Descriptions	1077
Table 23-1 Write Buffer Auxiliary Registers	1079
Table 23-2 WB_CTRL Register Field Description	1080
Table 23-3 WB_STATUS Register Field Description	1082
Table 23-4 WB_BUILD Register Field Description	1083
Table 24-1 BPU Registers	1087
Table 24-2 BPU_CTRL Field Description	1089
Table 24-3 BC_RAM_ADDR Field Description	1092
Table 24-4 LR and SR to BC_RAM_ADDR Description	1092
Table 24-5 BC_RAM_INFO Field Description	1094
Table 24-6 BPU Branch Types	1095
Table 24-7 LR and SR to BC_RAM_INFO Description	1096
Table 24-8 BC_RAM_BTA Field Description	1097
Table 24-9 LR and SR to BC_RAM_BTA Description	1097
Table 24-10 PT_RAM_ADDR Bit Field Description	1099
Table 24-11 LR and SR to PT_RAM_ADDR	1099
Table 24-12 PT_RAM_DATA Bit Field Description	1101
Table 24-13 LR and SR to PT_RAM_DATA	1102
Table 24-14 BPU_BUILD Field Descriptions	1103
Table 24-15 BPU Debug Auxiliary Registers	1105
Table 25-1 LPB_CTRL Field Description	1114
Table 25-2 LPB_BUILD Field Description	1115
Table 26-1 Host Debug Interface Auxiliary Registers	1117

Table 26-2 DEBUG Register Field Description	1118
Table 26-3 Internal Clock During Sleep Mode	1121
Table 26-4 DEBUGI Register Field Description	1122
Table 27-1 Auxiliary Registers for Actionpoints	1126
Table 27-2 State of Auxiliary Registers upon Reset	1128
Table 27-3 Actionpoint Build Field Descriptions	1130
Table 27-4 Actionpoint Target Field, AT, Type	1133
Table 27-5 Transaction Type Field, TT, Type	1134
Table 27-6 Mode Field, M, Type	1134
Table 27-7 Pair Field, P, Type	1135
Table 27-8 Actionpoint Action Field, AA Field, Type	1136
Table 27-9 Quad Type	1137
Table 27-10 Actionpoint Isolation Enable Field Descriptions	1159
Table 27-11 SMART_BUILD	1163
Table 27-12 SMART_CONTROL Register	1164
Table 27-13 SMART_DATA Register	1166
Table 27-14 SRC_ADDR Value	1167
Table 27-15 DEST_ADDR Register	1167
Table 27-16 FLAGS Value Register	1167
Table 27-17 Debug Register Addresses	1169
Table 27-18 DB_STATUS Field Description	1170
Table 27-19 DB_CMD Field Description	1171
Table 27-20 DB_ADDR Field Description	1172
Table 27-21 DB_DATA Field Description	1173
Table 27-22 DB_ADDR Field Description	1174
Table 28-1 Performance Auxiliary Registers	1178
Table 28-2 Countable-Conditions Registers	1179
Table 28-3 CC_INDEX Field Description	1180
Table 28-4 CC_NAME0 Field Description	1181

Table 28-5 CC_NAME1 Field Description	1182
Table 28-6 PCT_CONFIG Field Description	1187
Table 28-7 PCT_CONTROL Field Description	1188
Table 28-8 PCT_INDEX Field Description	1190
Table 28-9 PCT_UFLAGS Field Description	1194
Table 28-10 PCT_BUILD Field Description	1199
Table 28-11 CC_BUILD Field Description	1200
Table 29-1 Auxiliary Registers for Hardware Counter 0 (has_timer_0 == true)	1202
Table 29-2 Auxiliary Registers for Hardware Counter 1 (has_timer_1 == true)	1202
Table 29-3 Auxiliary Register for Real-time counter	1203
Table 29-4 TIMER_0 Field Descriptions	1205
Table 29-5 RTC Control Register	1211
Table 29-6 TIMER_BUILD Field Descriptions	1215
Table 30-1 Auxiliary Registers at the Cluster Level	1217
Table 30-2 Cluster Registers	1218
Table 30-3 CLUSTER_ID Register	1219
Table 30-4 GLOBAL_CLK_GATE_DIS Register	1220
Table 30-5 CLUSTER_VCCM_QoS Register	1222
Table 30-6 CC_NMI_STATUS Register	1224
Table 30-7 CC_NMI_CAUSE Register	1226
Table 30-8 NMI Cause Codes	1226
Table 30-9 CLUSTER_BUILD Register	1230
Table 31-1 I/O Coherency Programming Registers	1233
Table 31-2 BIU_IO_0_COH_ENABLE Bit Field Description	1234
Table 31-3 BIU_IO_0_COH_PARTIAL Bit Field Description	1235
Table 31-4 BIU_IO_0_COH_AP0_BASE Bit Field Description	1236
Table 31-5 BIU_IO_0_COH_AP0_SIZE Bit Field Description	1237
Table 31-6 BIU_BUILD Register	1238
Table 32-1 Power Domain Management Registers	1241

Table 32-2 PDM_PSTAT Field Description	1243
Table 32-3 RTT_PDM_PSTAT Field Description	1245
Table 32-4 PDM_PMODE Field Description	1246
Table 32-5 PDM_SCU_DIS_IND Field Description	1247
Table 32-6 PDM_CC_PMODE Field Description	1248
Table 32-7 PDM_CC_PSTAT Field Description	1249
Table 32-8 PDM_CC_PSTAT Field Description	1250
Table 32-9 PDM_CSM_PMODE Field Description	1252
Table 32-10 PDM_CSM_PSTAT Field Description	1253
Table 32-11 PDM_CSM_PSTAT Field Description	1254
Table 32-12 PDM_SLC_PMODE Field Description	1255
Table 32-13 PDM_SLC_PSTAT Field Description	1256
Table 32-14 PDM_CSM_P_PMODE Field Description	1257
Table 32-15 PDM_CSM_P_PSTAT Field Description	1259
Table 32-16 PDM_SLC_P_PMODE Field Description	1262
Table 32-17 PDM_SLC_P_PSTAT Field Description	1264
Table 32-18 PDM_CSM_SM_P_CFG Field Description	1267
Table 32-19 Legal Combination value for SM_CFG and PDM_CSM_SM_P_CFG	1267
Table 32-20 DVFS_CLUSTER_PL Field Description	1269
Table 32-21 Execution Control Registers	1269
Table 32-22 EXEC_CTRL Field Description	1270
Table 32-23 PDM_DVFS_BUILD Field Description	1271
Table 33-1 ARConnect Registers	1273
Table 33-2 CONNECT_CMD Register Field Description	1275
Table 33-3 CONNECT_WDATA Register Field Description	1276
Table 33-4 CONNECT_READBACK Register Field Description	1277
Table 33-5 CONNECT_SYSTEM_BUILD Register Field Description	1278
Table 33-6 CONNECT_SEMA_BUILD Register Field Description	1280
Table 33-7 CONNECT_MESSAGE_BUILD Register Field Description	1281

Table 33-8 CONNECT_PMU_BUILD Field Description	1283
Table 33-9 CONNECT_IDU_BUILD Field Description	1284
Table 33-10 CONNECT_GFRC_BUILD Field Description	1285
Table 33-11 CONNECT_IVC_BUILD Field Description	1286
Table 33-12 CONNECT_ICI_BUILD Field Description	1287
Table 33-13 CONNECT_ICD_BUILD Field Description	1288
Table 33-14 CONNECT_ASI_BUILD Field Description	1289
Table 33-15 CONNECT_PDM_BUILD Field Description	1290
Table 34-1 Cluster DMA Auxiliary Register List	1296
Table 34-2 DMA_C_CTRL_AUX Field Description	1298
Table 34-3 DMA_C_CHAN_AUX Field Description	1299
Table 34-4 DMA_C_SRC_AUX Field Description	1300
Table 34-5 DMA_C_SRC_HI_AUX Field Description	1301
Table 34-6 DMA_C_DST_AUX Field Description	1302
Table 34-7 DMA_C_DST_HI_AUX Field Description	1303
Table 34-8 DMA_C_ATTR_AUX Field Description	1304
Table 34-9 DMA_C_LEN_AUX Field Description	1306
Table 34-10 DMA_C_HANDLE_AUX Field Description	1307
Table 34-11 DMA_C_EVSTAT_AUX Field Description	1308
Table 34-12 DMA_C_EVSTAT_CLR_AUX Field Description	1309
Table 34-13 DMA_C_STATUS_AUX Field Description	1310
Table 34-14 DMA_C_INTSTAT_AUX Field Description	1311
Table 34-15 DMA_C_INTSTAT_CLR_AUX Field Description	1313
Table 34-16 DMA_C_ERRHANDLE_AUX Field Description	1314
Table 34-17 DMA_S_CTRL_AUX Field Description	1315
Table 34-18 DMA_S_DONESTATD_AUX Field Description	1316
Table 34-19 DMA_S_DONESTATD_AUX Field Description	1317
Table 34-20 DMA_S_STATC_AUX Field Description	1324
Table 34-21 DMA_BUILD Field Description	1327

Table 35-1 FPU Registers	1331
Table 35-2 FPU_CTRL Field Description	1332
Table 35-3 FPU_STATUS Field Description	1334
Table 35-4 FPU_BUILD Field Descriptions	1338
Table 37-1 RPU Indirect Programming Interface Registers	1347
Table 37-2 RPU_CMND Bit Field Description	1348
Table 37-3 RPU Programming Registers	1349
Table 37-4 RPU_BUILD Field Descriptions	1358
Table 38-1 16-Bit Signed 1Q15 Data Representation in a 32-bit Register	1362
Table 38-2 32-Bit Signed Fractional Data	1362
Table 38-3 DSP Auxiliary Registers	1362
Table 38-4 FMP_CTRL Field Descriptions	1363
Table 38-5 Build Configuration Registers	1363
Table 38-6 FMP_BUILD Field Descriptions	1364
Table 38-1 DSP Configuration Options	1422
Table 39-1 8-Bit Signed Integer Vector	1423
Table 39-2 16-Bit Signed Integer	1423
Table 39-3 16-Bit Unsigned Integer	1424
Table 39-4 16-Bit Signed Integer Vector	1424
Table 39-5 16-Bit Unsigned Integer Vector	1424
Table 39-6 16-Bit Signed Fractional Vector	1425
Table 39-7 Complex 16+16 Bit Complex Signed Fractional Data	1425
Table 39-8 32-Bit Signed Integer	1425
Table 39-9 32-Bit Unsigned Integer	1425
Table 39-10 32-Bit Signed Fractional Data	1426
Table 39-11 Accumulator Instructions	1428
Table 40-1 DSP Auxiliary Registers	1431
Table 40-2 ACC0_GLO Field Descriptions	1433
Table 40-3 ACC0_GHI Field Descriptions	1435

Table 40-4 DSP_FFT_CTRL Field Descriptions	1437
Table 40-5 DSP_CTRL Field Descriptions	1438
Table 40-6 DSP_BUILD Field Descriptions	1440
Table 41-1 AGU Registers	1444
Table 41-2 Number of Address Pointer Registers	1445
Table 41-3 Post-Incrementing Address Pointer Modes	1446
Table 41-4 AGU_AUX_APx Field Description	1447
Table 41-5 Number of Offset or Wrap Registers	1448
Table 41-6 Number of Modifier Registers	1449
Table 41-7 AGU_BUILD Field Descriptions	1452
Table 42-1 List of Instructions	1453
Table A-1 Baseline Differences: ARC 600, ARC 700, and ARCV2	1942
Table A-2 ISA Option Differences: ARC 600, ARC 700, and ARCV2	1944
Table A-3 ISA Features: ARC EM and ARC HS	1946
Table B-1 Implementation Behavior on Division Overflow	1950
Table C-1 CONNECT_READBACK Register Field Descriptions for CMD_CHECK_CORE_ID	1952
Table C-2 CONNECT_READBACK Register Field Descriptions for CMD_INTERRUPT_READ_STATUS ..	1955
Table C-3 CONNECT_READBACK Register Field Descriptions for CMD_INTRPT_CHECK_SOURCE ...	1956
Table C-4 CONNECT_READBACK Register Field Descriptions for CMD_SEMA_CLAIM_AND_READ ...	1960
Table C-5 CONNECT_WDATA Register Field Description for CMD_MSG_SET_ECC_CTRL	1968
Table C-6 CONNECT_READBACK Register Field Description for CMD_MSG_READ_ECC_SBE_CNT ...	1970
Table C-7 CONNECT_WDATA Field Descriptions for CMD_DEBUG_RESET	1971
Table C-8 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_EN	1976
Table C-9 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_CORE	1978
Table C-10 CONNECT_WDATA Field Descriptions	1982
Table C-11 CONNECT_WDATA Register Field Descriptions for CMD_PDM_SET_PM	1986
Table C-12 CONNECT_READBACK Register Field Descriptions for CMD_PDM_READ_STATUS	1987
Table C-13 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MODE	1995
Table C-14 CONNECT_WDATA Register Field Description for CMD_IDU_SET_DEST	1996

Table C-15 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_STATUS 2000

Table C-16 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_SOURCE 2001

Table C-17 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MASK 2003

Table C-18 CONNECT_WDATA Register Field Description for CMD_IDU_CHECK_FIRST 2004

Preface

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 for the ARC HS family of processors.

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.

Document Organization

This document is organized into the following parts:

- [Quick Reference](#): Provide a list of all auxiliary registers, build configuration registers, and exceptions available in the ARCV2 ISA-based processor.
- [ARCV2 Baseline ISA](#): Describes the data organization, core and auxiliary register sets, instruction set summary, and ARCV2 baseline instructions available in the processor.
- [Memory and System Components](#): Describes the optional extensions to the ISA.
- [FastMath Extension Pack](#): Describes the ISA for the FastMath extension pack.
- [ARCV2 Scalar DSP Extensions](#): Describes the ARCV2 DSP ISA that includes DSP instructions for accelerating signal processing algorithms, including vector and complex MUL/MAC operations.
- [Appendices](#): Describe the ISA version differences across processors and implementation-dependent behavior.

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
6359-003	February 2023	Version based on the ARC HS 4.00b release
6359-002	September 2021	Version based on the ARC HS 4.10a release
6359-001	June 2019	Initial version is based on the ARC HS 3.50a release

Differences Between ARC HS3x and ARC HS4x Processors

Table 1-1 Differences Between ARC HS4x and ARC HS3x Processors

Feature		HS3x	HS4x
Non-maskable imprecise exception (IRQ_BUILD [30:29])		Not supported	Supported
Memory Protection Unit	MPU version	0x03	0x06
	32 MPU regions	Not supported	Supported
	MPU cacheability attributes	Not supported	Supported
	Volatile and strict ordering attributes	Not supported	Supported
Memory Management Unit	MMU version	0x04	0x05
	LRU algorithm	Pseudo	True
	Lock bit	Absent	Present in the descriptor register
	TLBResetLRU	Absent	Present
	ITLB_NUM_ENTRIES	Fixed to 4	4, 8, 16
	DTLB_NUM_ENTRIES	Fixed to 8	8, 16
Coexistence of MMU and MPU in the same configuration		Not supported	Supported
Performance counters for L1 and L2 cache miss		Not supported	Supported
Performance counters for SYNC, DSYNC, DMB instructions		Not supported	Supported
Dual issue (Execution Rate Control Register , EXEC_CTRL)		Not supported	Supported
ROM Patching Unit		Not supported	Supported
ARCV2 Scalar DSP Extensions		Not supported	Supported

Table 1-1 Differences Between ARC HS4x and ARC HS3x Processors (Continued)

Address Generation Unit Auxiliary Interface		Not supported	Supported
Instruction Cache	Version (I_CACHE_BUILD)	0x04	0x06
	Waylock mechanism for Instruction cache (IC_WAYLOCK)	Not supported	Supported
	Prefetch line in instruction cache	Not supported	Supported
	Prefetch Region in instruction cache	Not supported	Supported
Loop Buffer		Not supported	Supported
Write Buffer		Not supported	Supported
EDC protection on pipeline registers (ERP_BUILD [23])		Not supported	Supported
Cluster DMA Controller		Not supported	Supported
Byte order (-byte_order)		little, big	little
FusedMultiplyAdd instructions		Configurable	Not configurable (-fpu_fma_option) is deprecated. These instructions are available by default.
Half-precision floating-point formats		Not supported	Supported
Half-precision to Single-precision conversion instructions (use -fpu_hp_option)		Not supported	Supported
Secure debug access		Not supported	Supported

Synopsys Statement on Inclusivity and Diversity

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

Quick Reference

In this part:

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

1

Build and Auxiliary Register List

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

Table 1-1 Auxiliary Register List

Register Address	Register Name	Power Domain
0x02	Loop Start Register, LP_START	PD1
0x03	Loop End Register, LP_END	PD1
0x04	Core Identity Register, IDENTITY	PD1
0x05	Debug Register, DEBUG	PD1
0x06	Program Counter, PC	PD1
0x07	Upper Physical Memory Address, Register, MEMSEG	PD1
0x08	Execution Rate Control Register, EXEC_CTRL	PD1
0x0A	Status Register, STATUS32	PD1
0x0B	Status Register Priority 0, STATUS32_P0	PD1
0x0D	Saved User Stack Pointer, AUX_USER_SP	PD1
0x0E	Interrupt Context Saving Control Register, AUX_IRQ_CTRL	PD1
0x0F	Interrupt and Register Bank Debug Register, DEBUGI	PD1
0x10	Invalidate Instruction Cache, IC_IVIC	PD1
0x11	Instruction Cache Control Register, IC_CTRL	PD1
0x13	Lock Instruction Cache Line, IC_LIL	PD1
0x14	Instruction Cache Way-Lock Register, IC_WAYLOCK	PD1
0x16	Invalidate Instruction-Cache Start Region, IC_IVIR	PD1
0x17	Invalidate Instruction-Cache End Region, IC_ENDR	PD1
0x18	DCCM Base Address, AUX_DCCM	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x19	Invalidate Instruction-Cache Line, IC_IVIL	PD1
0x1A	Instruction Cache External-Access Address, IC_RAM_ADDR	PD1
0x1B	Instruction-Cache Tag Access, IC_TAG	PD1
0x1D	Instruction Cache Data Access, IC_DATA	PD1
0x1E	Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG	PD1
0x1F	Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG_HI	PD1
0x21	Timer 0 Count Register, COUNT0	PD1
0x22	Timer 0 Control Register, CONTROL0	PD1
0x23	Timer 0 Limit Register, LIMIT0	PD1
0x25	Interrupt Vector Base Register, INT_VECTOR_BASE	PD1
0x3C	ECC SBE Counter Register, ECC_SBE_COUNT	PD1
0x3F	Error Protection Hardware Control Register, ERP_CTRL	PD1
0x43	Active Interrupts Register, AUX_IRQ_ACT	PD1
0x47	Invalidate Data Cache, DC_IVDC	PD1
0x48	Data Cache Control Register, DC_CTRL	PD1
0x49	Lock Data Cache Line, DC_LDL	PD1
0x4A	Invalidate Data Line, DC_IVDL	PD1
0x4B	Flush Data Cache, DC_FLSH	PD1
0x4C	Flush Data Line, DC_FLDL	PD1
0x4D	Data-Cache Region Start Address, DC_STARTR	PD1
0x4E	Data-Cache Region End Address, DC_ENDR	PD1
0x58	Data Cache External Access Address, DC_RAM_ADDR	PD1
0x59	Data Cache Tag Access, DC_TAG	PD1
0x5B	Data Cache Data Access, DC_DATA	PD1
0x5E	AUX_VOLATILE	PD1
0x5F	Data Cache External Access RAM Address Physical Tag Format, DC_PTAG_HI	PD1
0x60 -0x7F	Build Configuration Registers	PD1
0xC0 -0xFF		
0x100	Timer 1 Count Register, COUNT1	PD1
0x101	Timer 1 Control Register, CONTROL1	PD1
0x102	Timer 1 Limit Register, LIMIT1	PD1
0x103	RTC Control Register, AUX_RTC_CTRL	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x104	RTC Count Low Register, AUX_RTC_LOW	PD1
0x105	RTC Count High Register, AUX_RTC_HIGH	PD1
0x200	Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING	PD1
0x201	Software Interrupt Trigger, AUX_IRQ_HINT	PD1
0x206	Interrupt Priority Register, IRQ_PRIORITY	PD1
0x208	ICCM base address, AUX_ICCM	PD1
0x20A	Peripheral Memory Region, DMP_PER_AUX	PD1
0x20B	Data Memory Attribute Register, DATA_MEM_ATTR	PD1
0x20C	Write Buffer Control Register, WB_CTRL	PD1
0x20D	Write Buffer Status Register, WB_STATUS	PD1
0x220	Actionpoint Match Value, AP_AMV0	PD1
0x221	Actionpoint Match Mask, AP_AMM0	PD1
0x222	Actionpoint Control, AP_AC0	PD1
0x223	Actionpoint Match Value, AP_AMV1	PD1
0x224	Actionpoint Match Mask, AP_AMM1	PD1
0x225	Actionpoint Control, AP_AC1	PD1
0x226	Actionpoint Match Value, AP_AMV2	PD1
0x227	Actionpoint Match Mask, AP_AMM2	PD1
0x228	Actionpoint Control, AP_AC2	PD1
0x229	Actionpoint Match Value, AP_AMV3	PD1
0x22A	Actionpoint Match Mask, AP_AMM3	PD1
0x22B	Actionpoint Control, AP_AC3	PD1
0x22C	Actionpoint Match Value, AP_AMV4	PD1
0x22D	Actionpoint Match Mask, AP_AMM4	PD1
0x22E	Actionpoint Control, AP_AC4	PD1
0x22F	Actionpoint Match Value, AP_AMV5	PD1
0x230	Actionpoint Match Mask, AP_AMM5	PD1
0x231	Actionpoint Control, AP_AC5	PD1
0x232	Actionpoint Match Value, AP_AMV6	PD1
0x233	Actionpoint Match Mask, AP_AMM6	PD1
0x234	Actionpoint Control, AP_AC6	PD1
0x235	Actionpoint Match Value, AP_AMV7	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x236	Actionpoint Match Mask, AP_AMM7	PD1
0x237	Actionpoint Control, AP_AC7	PD1
0x23F	Watchpoint Program Counter, AP_WP_PC	PD1
0x240	Countable Conditions Index Register, CC_INDEX	PD1
0x241	Countable Conditions Name0 Register, CC_NAME0	PD1
0x242	Countable Conditions Name1 Register, CC_NAME1	PD1
0x250	Count-Value Registers, PCT_COUNTL	PD1
0x251	Count-Value Registers, PCT_COUNTH	PD1
0x252	Snapshot-Value Registers, PCT_SNAPL	PD1
0x253	Snapshot-Value Registers, PCT_SNAPH	PD1
0x254	Configuration Register, PCT_CONFIG	PD1
0x255	Control Register, PCT_CONTROL	PD1
0x256	Index-Select Register, PCT_INDEX	PD1
0x259	Address-Range Registers, PCT_RANGEL	PD1
0x25A	Address-Range Registers, PCT_RANGEH	PD1
0x25B	User-Flag Register, PCT_UFLAGS	PD1
0x25C	Interrupt Trigger Value, PCT_INT_CNTL	PD1
0x25D	Interrupt Trigger Value, PCT_INT_CNTH	PD1
0x25E	Interrupt Control Value, PCT_INT_CTRL	PD1
0x25F	Interrupt Active, PCT_INT_ACT	PD1
0x260	User Stack Region Top Address, USTACK_TOP	PD1
0x261	User Stack Region Base Address, USTACK_BASE	PD1
0x264	Kernel Stack Region Top Address, KSTACK_TOP	PD1
0x265	Kernel Stack Region Base Address, KSTACK_BASE	PD1
0x290	Jump and Link Indexed Base Address, JLI_BASE	PD1
0x291	Load Indexed Base Address, LDI_BASE	PD1
0x292	Execute Indexed Base Address, EI_BASE	PD1
0x298	CLUSTER ID Register, CLUSTER_ID	PD1
0x300	Floating-Point Unit Control Register, FPU_CTRL	PD1
0x301	Floating-Point Unit Status Register, FPU_STATUS	PD1
0x390	RPU Command Register, RPU_CMND	PD1
0x391	RPU Address Register, RPU_ADDR	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x392	RW	PD1
0x400	Exception Return Address, ERET	PD1
0x401	Exception Return Branch Target Address, ERBTA	PD1
0x402	Exception Return Status, ERSTATUS	PD1
0x403	Exception Cause Register, ECR	PD1
0x404	Exception Fault Address, EFA	PD1
0x409	MPU Enable Register, MPU_EN	PD1
0x40A	Interrupt Cause Registers, ICAUSE	PD1
0x40B	Interrupt Select, IRQ_SELECT	PD1
0x40C	Interrupt Enable Register, IRQ_ENABLE	PD1
0x40D	Interrupt Trigger Register, IRQ_TRIGGER	PD1
0x40F	Interrupt Status Register, IRQ_STATUS	PD1
0x410	User Mode Extension Enable Register, XPU	PD1
0x412	Branch Target Address, BTA	PD1
0x415	Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL	PD1
0x416	Interrupt Pending Register, IRQ_PENDING	PD1
0x420	MPU Exception Cause Register, MPU_ECR	PD1
0x422	MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB31	PD1
0x423	MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP31	PD1
0x424 to 0x441	Memory Protection Unit Descriptor Base and Permission Registers	PD1
0x44F	User Extension Flags Register, XFLAGS	PD1
0x460	TLB Page Descriptor Register for 32-Bit Virtual Address, TLBPD0 (MMU)	PD1
0x461	TLB Page Descriptor Register for 32-Bit Physical Address, TLBPD1 (MMU)	PD1
0x463	TLB Page Descriptor Register for 64-Bit Physical Address, TLBPD1_HI (MMU)	PD1
0x464	TLB Index Register, TLBIndex (MMU)	PD1
0x465	TLB Command Register, TLBCommand (MMU)	PD1
0x468	Process Identity Register, PID (MMU)	PD1
0x46A	Shared Library Address Space Identifier Register, SASID0, SASID1 (MMU)	PD1
0x46B	Shared Library Address Space Identifier Register, SASID0, SASID1 (MMU)	PD1
0x46C	Scratch Data Register, SCRATCH_DATA0 (MMU)	PD1
0x470	FMP Control Register, FMP_CTRL	PD1
0x480	Flush and Initialize Branch Predictor Register, BPU_FLUSH	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x481	BPU Control Register, BPU_CTRL	PD1
0x488	Loop Buffer Control, LPB_CTRL	PD1
0x500	I/O Coherency Enable Register, BIU_IO_0_COH_ENABLE	PD1
0x501	I/O Coherency Partial Write Register, BIU_IO_0_COH_PARTIAL	PD1
0x508	I/O Coherency Address Aperture Base Address, BIU_IO_0_COH_AP0_BASE	PD1
0x509	I/O Coherency Aperture Size, BIU_IO_0_COH_AP0_SIZE	PD1
0x580	Accumulator Low Register, ACC0_LO	PD1
0x581	Low Accumulator Guard and Status Register, ACC0_GLO	PD1
0x582	Accumulator High Register, ACC0_HI	PD1
0x583	High Accumulator Guard and Status Register, ACC0_GHI	PD1
0x598	DSP Butterfly Instructions Data Register, DSP_BFLY0	PD1
0x59E	DSP FFT Control Register, DSP_FFT_CTRL	PD1
0x59F	DSP Control Register, DSP_CTRL	PD1
0x5C0 to 0x5CF	AGU Address Pointer Registers, AGU_AUX_APx	PD1
0x5D0 to 0x5DF	AGU Offset Registers, AGU_AUX_OSx	PD1
0x5E0 to 0x5F7	AGU Modifier Registers, AGU_AUX_MODx	PD1
0x600	ARConnect Command Register, CONNECT_CMD	PD1
0x601	ARConnect Write Data Register, CONNECT_WDATA	PD1
0x602	ARConnect Read Data Register, CONNECT_READBACK	PD1
0x610	Core Power Status Register, PDM_PSTAT	Always on
0x611	ARC Trace Power Status Register, RTT_PDM_PSTAT	Always on
0x613	Power Down Register, PDM_PMODE	Always on
0x614	Core Snoop Indicator Control Register, PDM_SCU_DIS_IND	Always on
0x620	Power Down Cluster Register, PDM_CC_PMODE	Always on
0x621	Cluster Power Status Register, PDM_CC_PSTAT	Always on
0x622	Power Down Cluster Shared Memory Register, PDM_CSM_PMODE	Always on
0x623	Cluster CSM Power Status Register, PDM_CSM_PSTAT	Always on
0x624	Power Down L2 Cache Register, PDM_SLC_PMODE	Always on

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x625	Cluster L2 Cache Power Status Register, PDM_SLC_PSTAT	Always on
0x626	Cluster Performance Register, DVFS_CLUSTER_PL	Always on
0x627	Partial Power-Down Cluster Shared Memory Register, PDM_CSM_P_PMODE	Always on
0x628	Partial Power-Down Cluster Shared Memory Register, PDM_CSM_P_PSTAT	Always on
0x629	Partial Power-Down L2 Cache Register, PDM_SLC_P_PMODE	Always on
0x62A	Partial Power-Down L2 Cache Register, PDM_SLC_P_PSTAT	Always on
0x62B	Shared Memory Power Down Register, PDM_CSM_SM_P_CFG	Always on
0x700	SmaRT Control Register, SMART_CONTROL	PD1
0x701	SmaRT Data Register, SMART_DATA	PD1
0x901	L2 Cache Configuration Register, SLC_AUX_CACHE_CONFIG	PD1
0x902	L2 Cache ECC Configuration Register, SLC_ECC_CONFIG	PD1
0x903	L2 Cache Control Register, SLC_AUX_CACHE_CTRL	PD1
0x904	L2 Cache Flush Register, SLC_AUX_CACHE_FLUSH	PD1
0x905	L2 Cache Invalidate Register, SLC_AUX_CACHE_INV	PD1
0x906 -0x90D	L2 Cache Way-Lock Register, SLC_AUX_CACHE_LOCK n	PD1
0x90E	L2 Cache Line Lock Register, SLC_AUX_LINE_LOCK	PD1
0x90F	L2 Cache Line Lock High Register, SLC_AUX_LINE_LOCK1	PD1
0x910	L2 Cache Line Invalidate Register, SLC_AUX_LINE_INV	PD1
0x911	L2 Cache Line Invalidate High Address Register, SLC_AUX_LINE_INV1	PD1
0x912	L2 Cache Line Flush Register, SLC_AUX_LINE_FLUSH	PD1
0x913	L2 Cache Line Flush High Address Register, SLC_AUX_LINE_FLUSH1	PD1
0x914	L2 Cache Line Operation Start Address Register, SLC_AUX_RGN_START	PD1
0x915	L2 Cache Line Operation Start High Address Register, SLC_AUX_RGN_START1	PD1
0x916	L2 Cache Line Operation End Address Register, SLC_AUX_RGN_END	PD1
0x917	L2 Cache Line Operation End High Address Register, SLC_AUX_RGN_END1	PD1
0x918	L2 Cache RAM Access Register, SLC_AUX_LINE_ADDR	PD1
0x919	L2 Cache Line RAM Access High Address Register, SLC_AUX_LINE_ADDR1	PD1
0x91A	L2 Cache Direct Index Register, SLC_AUX_DIRECT_IDX	PD1
0x91B	L2 Cache TAG Register, SLC_AUX_TAG_DATA	PD1
0x91C	L2 Cache TAG High Address Register, SLC_AUX_TAG_DATA1	PD1
0x91D	L2 Cache Status Register, SLC_AUX_STATUS_DATA	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x91F	L2 Cache Data Register, SLC_AUX_DATA0 Register	PD1
0x920	L2 Cache Data High Register, SLC_AUX_DATA1	PD1
0x921	L2 Cache Data2 Register, SLC_AUX_DATA2	PD1
0x922	L2 Cache Data3 Register, SLC_AUX_DATA3	PD1
0x923	L2 Cache Fault Address Register, SLC_AUX_FAULT_ADDR	PD1
0x924	L2 Cache Fault Address High Register, SLC_AUX_FAULT_ADDR1	PD1
0x925	L2 Cache Fault Status Register, SLC_AUX_FAULT_STAT	PD1
0x926	L2 Cache Performance Monitor Command Register, SLC_AUX_PM_CMD	PD1
0x927	L2 Cache Performance Monitor Event Register, SLC_AUX_PM_EVENT	PD1
0x928	L2 Cache Performance Counter Overflow Register, SLC_AUX_PM_OVF	PD1
0x929	L2 Cache Performance Monitor Count0 Register, SLC_AUX_PM_CNT0	PD1
0x92A	L2 Cache Performance Monitor Count1 Register, SLC_AUX_PM_CNT1	PD1
0x92E -0x936	L2 Cache Way-Lock Register, SLC_AUX_CACHE_LOCK n	PD1
0x93C	L2 Cache SBE Counter Register, SLC_AUX_ECC_SBE_CNT	PD1
0x940	Cluster NMI Status Register, CC_NMI_STATUS	PD1
0x941	Cluster NMI Cause Register, CC_NMI_CAUSE	PD1
0x9A0	CSM Enable Register, CSM_ENABLE	Always on
0x9A1	CSM ECC Control Register, CSM_ECC_CTRL	PD1
0x9A8	Shared Memory Configuration Register, SM_CONFIG_REG	Always on
DMA_AUX_BASE +0x0	DMA Client Control Register, DMA_C_CTRL_AUX	PD1
DMA_AUX_BASE +0x1	DMA Client Channel Select Register, DMA_C_CHAN_AUX	PD1
DMA_AUX_BASE +0x2	DMA Client Source Address Register, DMA_C_SRC_AUX	PD1
DMA_AUX_BASE +0x3	DMA Client Source High Address Register, DMA_C_SRC_HI_AUX	PD1
DMA_AUX_BASE +0x4	DMA Client Destination Address Register, DMA_C_DST_AUX	PD1
DMA_AUX_BASE +0x5	DMA Client Destination High Address Register, DMA_C_DST_HI_AUX	PD1
DMA_AUX_BASE +0x6	DMA Client Attributes Register, DMA_C_ATTR_AUX	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
DMA_AUX_BASE +0x7	DMA Client Length Register, DMA_C_LEN_AUX	PD1
DMA_AUX_BASE +0x8	DMA Client Handle Register, DMA_C_HANDLE_AUX	PD1
DMA_AUX_BASE +0xA	DMA Event Status Register, DMA_C_EVSTAT_AUX	PD1
DMA_AUX_BASE +0xB	DMA Client Clear Event Status Register, DMA_C_EVSTAT_CLR_AUX	PD1
DMA_AUX_BASE +0xC	DMA Client Status Register, DMA_C_STATUS_AUX	PD1
DMA_AUX_BASE +0xD	DMA Client Interrupt Status Register, DMA_C_INTSTAT_AUX	PD1
DMA_AUX_BASE +0xE	DMA Client Clear Interrupt Status Register, DMA_C_INTSTAT_CLR_AUX	PD1
DMA_AUX_BASE +0xF	DMA Client Error Handle Register, DMA_C_ERRHANDLE_AUX	PD1
DMA_AUX_BASE +0x10	DMA Server Control Register, DMA_S_CTRL_AUX	PD1
DMA_AUX_BASE +0x20+D	DMA Server Done Status Register, DMA_S_DONESTATD_AUX	PD1
DMA_AUX_BASE +0x40+D	DMA Server Clear Done Status Register, DMA_S_DONESTATD_CLR_AUX	PD1
DMA_AUX_BASE +0x80+(C*8)	DMA Server Channel Tail Pointer Register, DMA_S_TAILC_AUX	PD1
DMA_AUX_BASE +0x81+(C*8)	DMA Server Channel Middle Pointer Register, DMA_S_MIDC_AUX	PD1
DMA_AUX_BASE +0x82+(C*8)	DMA Server Channel Head Pointer Register, DMA_S_HEADC_AUX	PD1
DMA_AUX_BASE +0x83+(C*8)	DMA Server Channel Base Register, DMA_S_BASEC_AUX	PD1
DMA_AUX_BASE +0x84+(C*8)	DMA Server Channel Last Register, DMA_S_LASTC_AUX	PD1
DMA_AUX_BASE +0x85+(C*8)	DMA Channel Priority Register, DMA_S_PRIORC_AUX	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
DMA_AUX_BASE +0x86+(C*8)	DMA Channel Control Register, DMA_S_STATC_AUX	PD1

1.1 Build Configuration Registers

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

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

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

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

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

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

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

Table 1-2 Build Configuration Registers

Number	Name	Power Domain
0x60	Build Configuration Registers Version, BCR_VER	PD1
0x63	BTA Configuration Register, BTA_LINK_BUILD	PD1
0x68	Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD	PD1
0x6D	Memory Protection Build Configuration Register, MPU_BUILD	PD1
0x6E	Core Register File Configuration Register, RF_BUILD	PD1
0x6F	MMU Build Configuration Register, MMU_BUILD	PD1
0x72	Data Cache Configuration Register, D_CACHE_BUILD	PD1

Table 1-2 Build Configuration Registers (Continued)

Number	Name	Power Domain
0x74	DCCM Configuration Register, DCCM_BUILD	PD1
0x75	Timers Configuration Register, TIMER_BUILD	PD1
0x76	Actionpoints Configuration Register, AP_BUILD	PD1
0x77	Instruction Cache Configuration Register, I_CACHE_BUILD	PD1
0x78	ICCM Configuration Register, ICCM_BUILD	PD1
0x7A	DSP Build Configuration Register, DSP_BUILD	PD1
0x7B	Multiplier Configuration Register, MULTIPLY_BUILD	PD1
0x7C	Swap Instruction Configuration Register, SWAP_BUILD	PD1
0x7D	Normalize Instruction Configuration Register, NORM_BUILD	PD1
0x7E	Min/Max Instruction Configuration Register, MINMAX_BUILD	PD1
0x7F	Barrel Shifter Configuration Register, BARREL_BUILD	PD1
0xC0	Branch Prediction Unit Configuration Register, BPU_BUILD	PD1
0xC1	Instruction Set Configuration Register, ISA_CONFIG	PD1
0xC3	Peripheral Region Build Configuration Register, DMP_PP_BUILD	PD1
0xC5	Stack Region Configuration Register, STACK_REGION_BUILD	PD1
0xC7	Error Protection Configuration Register, ERP_BUILD	PD1
0xC8	Floating-Point Unit Build Register, FPU_BUILD	PD1
0xCC	AGU Build Configuration Register, AGU_BUILD	PD1
0xCE	L2 Cache Build Configuration Register, SLC_BUILD	PD1
0xCF	CLUSTER Build Configuration Register, CLUSTER_BUILD	PD1
0xD0	ARConnect Build Configuration Register, CONNECT_SYSTEM_BUILD	PD1
0xD1	Inter-core Semaphore Unit BCR, CONNECT_SEMA_BUILD	PD1
0xD2	Inter-core Message Unit BCR, CONNECT_MESSAGE_BUILD	PD1
0xD3	Power Management Unit BCR, CONNECT_PMU_BUILD	PD1
0xD5	Interrupt Distribution Unit BCR, CONNECT_IDU_BUILD	PD1
0xD6	ARConnect Global Free Running Counter BCR, CONNECT_GFRC_BUILD	PD1
0xE0	Inter-Core Interrupt Unit BCR, CONNECT_ICI_BUILD	PD1
0xE1	Inter-Core Debug Unit BCR, CONNECT_ICD_BUILD	PD1

Table 1-2 Build Configuration Registers (Continued)

Number	Name	Power Domain
0xE2		PD1
0xE3	ARConnect PDM BCR, CONNECT_PDM_BUILD	PD1
0xE6	DMA Build Configuration Register, DMA_BUILD	PD1
0xE8	FMP Build Configuration Register, FMP_BUILD	PD1
0xE9	Loop Buffer Configuration Register, LPB_BUILD	PD1
0xE5	CSM Build Configuration Register, CSM_BUILD	PD1
0xF3	Interrupt Build Configuration Register, IRQ_BUILD	PD1
0xF5	Performance Counter Build-Configuration Register, PCT_BUILD	PD1
0xF6	Countable Conditions Build Configuration Register, CC_BUILD	PD1
0xF7	Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD	PD1
0xF8	ARCV2 ISA Profile Register, ISA_PROFILE	PD1
0xF9	Micro Architecture Configuration Register, MICRO_ARCH_BUILD	PD1
0xFC	RPU Build Configuration Register, RPU_BUILD	PD1
0xFD	Write Buffer BCR Register, WB_BUILD	PD1
0xFF	SMART_BUILD Configuration Register, SMART_BUILD	PD1

2

Exceptions List



Note

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

Table 2-1 Exception Vectors and Cause Codes

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Reset	Reset	0x00	0x00	0x00	0x00	0x000000
Bus Error from Instruction Memory	Memory Error	0x04	0x01	0x00	0x0u	0x01000u(a)
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
Bus Error from Data Memory	Memory Error	0x04	0x01	0x10	0x0u	0x01100u(a)'(b)
Data access from unpopulated ICCM or DCCM region	Memory Error	0x04	0x01	0x11	0x00	0x011100

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Data access spanning multiple data memory targets	Memory Error	0x04	0x01	0x12	0x00	0x011200
NVM write or erase programming failure	Memory Error	0x04	0x01	0x13	0x00	0x011300
NVM address out of range (instruction fetch or data access) only when NVM is configured	Memory Error	0x04	0x01	0x14	0x00	0x011400
XY access to illegal target memory	Memory Error	0x04	0x01	0x21	0x00	0x012100
XY access spanning multiple targets	Memory Error	0x04	0x01	0x22	0x00	0x012200
XY 64-bit destination is not 32-bit aligned	Memory Error	0x04	0x01	0x23	0x00	0x012300
Illegal Instruction	Instruction Error	0x08	0x02	0x00	0x00	0x020000
Illegal Instruction Sequence	Instruction Error	0x08	0x02	0x01	0x00	0x020100
Double Fault	EV_MachineCheck	0x0C	0x03	0x00	0x00	0x030000
Overlapping TLB Entries	EV_MachineCheck	0x0C	0x03	0x01	0x00	0x030100
Fatal TLB Error	EV_MachineCheck	0x0C	0x03	0x02	0x00	0x030200
Fatal Cache Error	EV_MachineCheck	0x0C	0x03	0x03	0x00	0x030300
Internal Memory Error on Instruction Fetch	EV_MachineCheck	0x0C	0x03	0x04	0xrr	0x0304rr(e)
Internal Memory Error on Data Access	EV_MachineCheck	0x0C	0x03	0x05	0xrr	0x0305rr(e)
Illegal Overlapping MPU Entries	EV_MachineCheck	0x0C	0x03	0x06	0x00	0x030600
Illegal Overlapping MPU entries (jump and branch target)	EV_MachineCheck	0x0C	0x03	0x06	0x01	0x030601(f)

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Non-Maskable imprecise exception	EV_MachineCheck	0x0C	0x03	0x14	0x00	0x031400
Non-maskable imprecise exception double fault	EV_MachineCheck	0x0C	0x03	0x14	0x01	0x031401
Instruction Fetch TLB Miss	EV_ITLBMiss	0x10	0x04	0x00	0x00	0x040000
Data TLB Miss on read	EV_DTLBMiss	0x14	0x05	0x01	0x00	0x050100
Data TLB miss on write	EV_DTLBMiss	0x14	0x05	0x02	0x00	0x050200
Data TLB miss on RMW	EV_DTLBMiss	0x14	0x05	0x03	0x00	0x050300
Instruction Fetch Protection Violation in MPU	EV_ProtV	0x18	0x06	0x00	0x04	0x060004
Instruction Fetch Protection Violation in MMU(d)	EV_ProtV	0x18	0x06	0x00	0x08	0x060008
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 Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in code protection scheme (parameter code 0x01)	EV_ProtV	0x18	0x06	0x02	0x01	0x060201

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in stack checking scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x02	0x02	0x060202
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in MPU (parameter code 0x04)	EV_ProtV	0x18	0x06	0x02	0x04	0x060204
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in 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 Read-Modify-Write (EX) protection violation in code protection (parameter code 0x01)	EV_ProtV	0x18	0x06	0x03	0x01	0x060301
Memory Read-Modify-Write (EX) protection violation in stack checking protection scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x03	0x02	0x060302
Memory Read-Modify-Write (EX) protection violation in MPU scheme (parameter code 0x04)	EV_ProtV	0x18	0x06	0x03	0x04	0x060304
Memory Read-Modify-Write (EX) protection violation in 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
Action Point Hit, Instruction Fetch	EV_PrivilegeV	0x1C	0x07	0x02	0xnn	0x0702nn(c)

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Privilege Violation	EV_PrivilegeV	0x1C	0x07	0x00	0x00	0x070000
Disabled Extension	EV_PrivilegeV	0x1C	0x07	0x01	0xnn	0x0701nn(c)
Action Point Hit, Memory or Register	EV_PrivilegeV	0x1C	0x07	0x02	0xnn	0x0702nn(c)
Software Interrupt	EV_SWI	0x20	0x08	0x00	0xnn	0x0800nn(c)
Trap	EV_Trap	0x24	0x09	0x00	0xnn	0x0900nn(c)
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 data access: a LD or ST using an unaligned AGU based access straddles a modulo wrap boundary	EV_Misaligned	0x34	0x0D	0x10	0x00	0x0D1000
Misaligned vector memory access	EV_VecUnit	0x38	0x0E	0x02	0x00	0x0E0200
Vector memory address out of range	EV_VecUnit	0x38	0x0E	0x03	0x00	0x0E0300
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 247.
- (c) An example of an extension is floating-point extension. For this extension, the following are the values of sub cause code and parameter:

mm = 0 — indicates floating-point extension exceptions

nn—indicates specific exception(s) raised

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

- (d) The rules for prioritizing exceptions are applied normally, when multiple MMU exceptions are raised on two adjacent pages accessed by non-aligned memory operations. That means that the highest priority exceptions are always taken in preference to lower priority exceptions. If two exceptions of the same type occur, for different pages, from the same memory reference, then the exception triggered by the effective byte address (EBA) of the memory operation should be taken first. If a reference spans two pages, then the second address will be EBA+4.
- (e) The parameter codes (rr) are implementation dependent. See the processor databook for more information about the error parameter codes for this exception.

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](#)

2

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.

2.1 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.

2.2 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 writeback
 - Pre- and Post-Address Register writeback
 - 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
- Multiprocessing support
 - ❑ Data synchronization and memory barrier instructions

2.3 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.

2.3.1 Core Register Set

ARCV2-based processors provide a set of general-purpose registers, allowing instructions to have up to two source operands and one destination. Programmer may use the general purpose registers (r0-r25, and r30) for any purpose. Some of the core registers have a defined purpose such as stack pointers, link registers, and loop counters. See section [“Core Register Set”](#) on page 125.

Additional banks of registers may be configured for fast interrupt response and context switching. See [“Banked Interrupt Registers”](#) on page 213.

2.3.2 Auxiliary Register Set

The auxiliary register set contains status and control registers, which by default are 32 bits wide. These auxiliary registers occupy a separate 32-bit address space from the normal memory-access (that is, load and store) instructions. Auxiliary registers accessed using distinct Load Register (LR), Store Register (SR), and Auxiliary EXchange (AEX) instructions. See [“Auxiliary Register Set”](#) on page 134.

2.3.3 32-bit Instruction Formats

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

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

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

2.3.4 16-bit Instruction Formats

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

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

2.3.5 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

2.4 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 2-1](#) describes the configuration options for the programming model supported by the ARCV2 architecture.



Note

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

2.4.1 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 2-1](#).

Table 2-1 ARCV2 Programming Model Configuration Options

Configuration Option	Value Range	Description
-byte_order	little, big	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.

Table 2-1 ARCV2 Programming Model Configuration Options (Continued)

Configuration Option	Value Range	Description
<code>-rgf_num_regs</code>	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.
<code>-rgf_banked_regs</code>	4, 8, 16, 32	Defines the number of registers replicated per register bank.
<code>-rgf_num_banks</code>	1 through 8	Defines the number of register file banks. Register bank 0 to Register bank 7. The ARCV2ISA supports up to 8 register banks. However, the processors might not implement all 8 register banks. See the processor databook for the number of register banks implemented in that processor.
<code>-intvbase_preset</code>	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).
<code>-has_interrupts</code>	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.
<code>-halt_on_reset</code>	false, true	Indicates whether the core is halted initially on reset. ■ false: core is not halted initially on reset. ■ true: core is halted initially on reset.
<code>-number_of_interrupts</code>	0 to 240 interrupts	Defines the number of interrupts in the interrupt controller. Note: This option is available only if <code>-has_interrupt==true</code> .
<code>-number_of_levels</code>	1 to 16	Defines the number of interrupt priority levels (0 to 15) in the interrupt controller. Note: This option is available only if <code>-has_interrupt==true</code> .
<code>-firq_option</code>	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 <code>-has_interrupt==true</code> .

Table 2-1 ARCV2 Programming Model Configuration Options (Continued)

Configuration Option	Value Range	Description
<code>-nmi_option</code>	0, 1, 2	Indicates whether the core includes an external signal using which you can trigger a non-maskable imprecise exception. 0: not supported 1: includes the non-maskable signal for a single-core 2: includes the non-maskable signal for multicore
<code>-vector_proc_profile</code>	false, true	A vector unit optionally configured with an ARC HS core displaces some instructions associated with the code density options. When a vector unit is configured in at least one ARC HS core within a multi-core cluster, all the cores within the cluster, including those without a vector unit, can maintain scalar ISA compatibility. This can be ensured by setting a cluster level option, <code>-vector_proc_profile==true</code> . See the <code>ISA_PROFILE</code> register for information that indicates how code density related op code groups are utilized in a given configuration.

2.4.2 Processor Component Options

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

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

2.4.3 ISA Options

[Table 2-2](#) 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 369.

Table 2-2 ARCV2 Instruction Set Options

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

Table 2-2 ARCV2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-shift_option	0	-	-
	1	ASR16, ASR8, LSR8, LSL8, ROL8, ROR8	
	2	Multi-bit shift or rotate operations: ASL, LSR, ASR, ROR, ASL_S, LSR_S, ASR_S. XBFU (when -bitfield_option ==true).	-
	3	ASR16, ASR8, LSR8, LSL8, ROL8, ROR8 Multi-bit shift or rotate operations: ASL, LSR, ASR, ROR, ASL_S, LSR_S, ASR_S. XBFU (when -bitfield_option ==true).	
-bitfield_option	false	-	
	true	XBFU (when SHIFT_OPTION is 2 or 3). Note: this option is enabled by default for cores with ARCOVER value of 0x50 or above.	
-os_opt_option	false	-	
	true	PREALLOC, SWI_S encoding with n6 operand, DSYNC, DMB	
-mpy_option	none	-	-
	mpy	MPYW, MPYUW	-
	mac (also referred to as mpy_option==7 in some places in the document)	All mpy_option=mpy instructions and MPY, MPYU, MPYM, MPYMU, MPY_S, MAC, MACU, DMPYH, DMPYHU, DMACH, DMACHU, VADD2H, VSUB2H, VADDSUB2H, VSUBADD2H	-

Table 2-2 ARCv2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
	mpyd (also referred to as mpy_option==8 in some places in the document)	All mpy_option=plus_dmpy instructions and the following: MPYD, MPYDU, MACD, MACDU, VMPY2H, VMPY2HU, VMAC2H, VMAC2HU	
	qmpy (also referred to as mpy_option==9 in some places in the document)	All mpy_option=plus_macd instructions and the following: QMPYH, QMPYHU, DMPYWH, DMPYWHU, QMACH, QMACHU, DMACWH, DMACWHU, VADD4H, VSUB4H, VADDSUB4H, VSUBADD4H, VADD2, VSUB2, VADDSUB, VSUBADD	
-atomic_option	false		-
	true	LLOCK, SCOND, WLFC LLOCKD and SCONDD instructions also require -ll64_option==true	-
-div_rem_option	none	-	DZ bit in STATUS32
	radix4_enhanced	DIV, DIVU, REM, REMU	
-ll64_option	false		
	true	LDD, STD LLOCKD and SCONDD instructions also require -atomic_option==true	
-has_fpu	false		
	true	FSMUL, FSADD, FSSUB, FSCMP, FSCMPF, FS2INT, FS2INT_RZ, FINT2S, FS2UINT, FS2UINT_RZ, FUINT2S, FSABS, FSNEG, FSMADD, FSMSUB	ACCH, ACCL for FSMADD and FSMSUB instructions

Table 2-2 ARCV2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Auxiliary Registers
-fpu_dp_option	false	FSMUL, FSADD, FSSUB, FSCMP, FSCMPF, FS2INT, FS2INT_RZ, FINT2S, FS2UINT, FS2UINT_RZ, FUINT2S, FSABS, FSNEG	
	true	All -fpu_dp_option ==false instructions and the following: FDMUL, FDADD, FDSUB, FDCMP, FDCMPF, FD2INT, FD2INT_RZ, FINT2D, FD2UINT, FD2UINT_RZ, FUINT2D, FD2L, FD2L_RZ, FL2D FD2UL, FD2UL_RZ, FUL2D, FS2L, FS2L_RZ, FL2S, FS2UL, FS2UL_RZ, FUL2S, FDABS, FDNEG, FDMADD, FDMSUB	ACCH, ACCL for FSMADD and FSMSUB instructions
-fpu_hp_option	false	-	
	true	FH2S, F2HS	
-fpu_div_option	false	-	
	true	if -fpu_dp_option ==false: FSDIV, FSSQRT if -fpu_dp_option ==true: FSDIV, FSSQRT, FDDIV, FDSQRT	
-has_dsp	true	See Table 42-1	

2.5 Custom Extensions

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

2.5.1 Extension Core Registers

The core register set has a total of 64 different addressable registers. Registers `r32` to `r57` `r32` to `r55` are available for extension purposes. [Figure 4-1](#) on page 125 shows the core register map.

2.5.2 Extension Auxiliary Registers

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

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

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

2.5.3 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 writeback to the core register set.

2.5.4 Extension Condition Codes

The condition code test on an instruction is encoded by using a 5-bit binary field which gives 32 different possible conditions that can be tested. The first 16 codes (`0x00-0x0F`) are the condition codes defined in the basecase version of ARCV2-based processor which use only the internal condition flags from the status register (`Z`, `N`, `C`, `V`). For more information, see [Table 6-7](#) on page 264.

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

3

Data Organization and Addressing

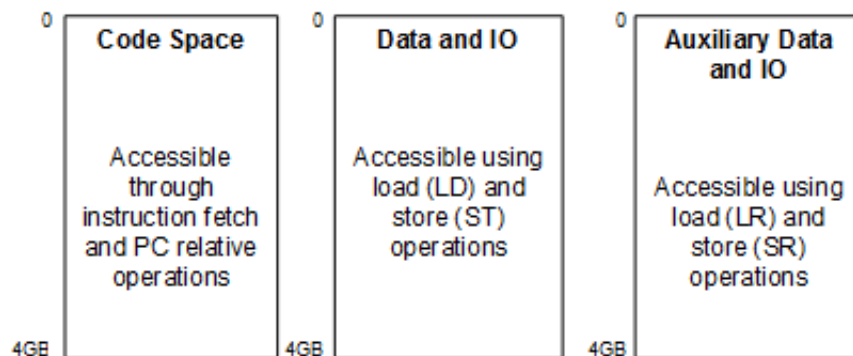
This chapter describes the data organization and addressing used by the ARCV2-based processor.

3.1 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 3-1](#) and [3.2](#) for more information.

Figure 3-1 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.

3.2 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 3-5](#) and [Figure 3-6](#).

3.2.1 Vector Operands

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

3.2.1.1 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 3-2 64-bit Element As a 64-bit Operand

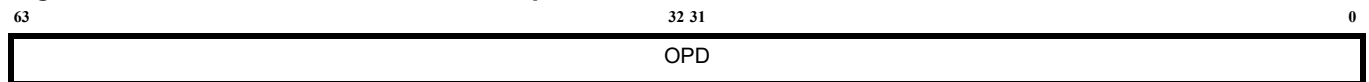


Figure 3-3 32-bit Elements in a 64-bit Operand

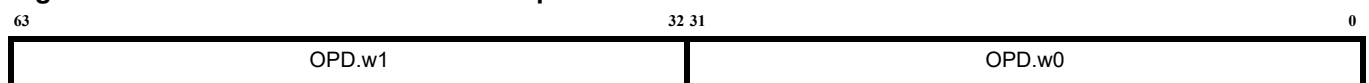
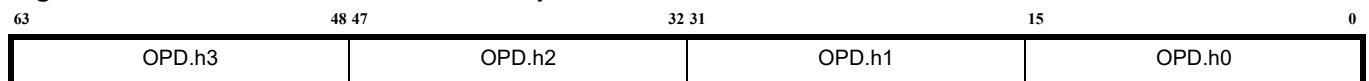


Figure 3-4 16-bit Elements in a 64-bit Operand



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 3-5 Register Pair Holding Operands in Little-endian Mode

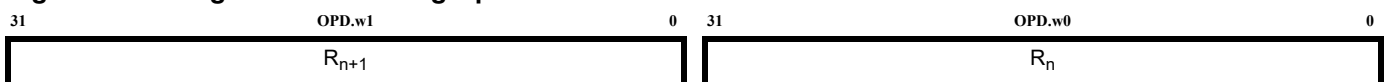
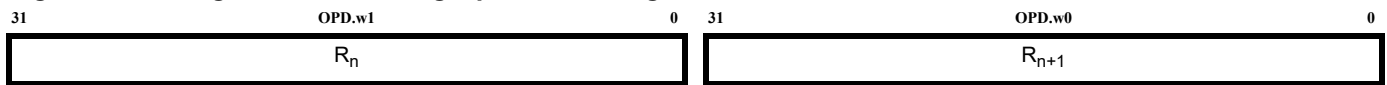


Figure 3-6 Register Pair Holding Operands in Big-endian Mode

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.

3.2.1.2 LDD and STD Operands

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

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

Consider the following double load instruction:

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

Little-endian mode (LE)

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

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

Big-endian mode (BE)

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

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

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

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

3.2.2 32-Bit Operand Data Elements

Figure 3-7 32-Bit Elements in a 32-bit Operand

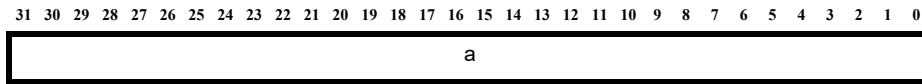
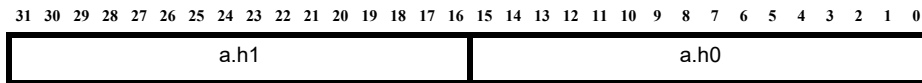


Figure 3-8 16-Bit Elements in a 32-bit Operand



3.2.3 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 3-9 and Figure 3-10 illustrate expansion of a `shimm` to a word (32 bits).

Figure 3-9 Expansion of a `u6` Literal to a 32-Bit Word (w-`shimm`)

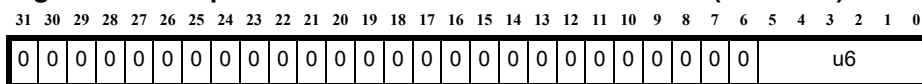


Figure 3-10 Expansion of an `s12` Literal to a 32-Bit Word (w-`shimm`)

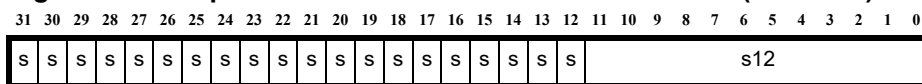


Figure 3-11 and Figure 3-12 illustrate expansion of a `shimm` to a half-word (16-bits).

Figure 3-11 Expansion of a `u6` Literal to Half-Word (16-bit, hw-`shimm`)

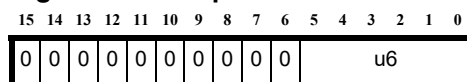
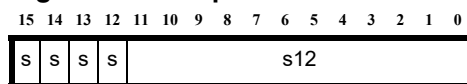


Figure 3-12 Expansion of an `s12` Literal to Half-Word (16-bit, hw-`shimm`)



3.2.4 Expansion of Literals in Vector Instructions

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

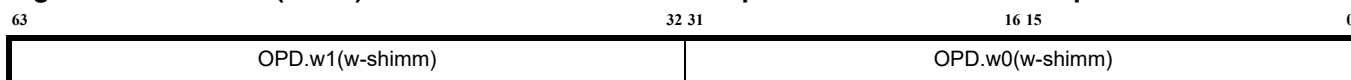
Figure 3-13 A Half-Word (16-bit) Short Limm Value Replicated to form a 64-Bit Operand

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

Where a `hw-shimm` represents `u6` or `s12` extended to half-word (16-bits). For an illustration of `hw-shimm`, see Figure 3-11 and Figure 3-12.

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

Figure 3-14 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 3-9](#) and [Figure 3-10](#).

Figure 3-15 and Figure 3-16 illustrate the expansion of two hw-shimm(16 bits) to a word (32 bits).

Figure 3-15 Two 16-Bit Vectors with Expanded u6 Operands

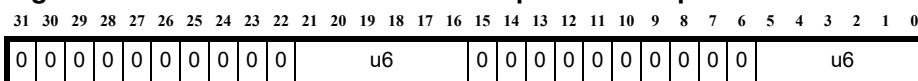


Figure 3-16 Two 16-Bit Vectors with Expanded s12 Operands

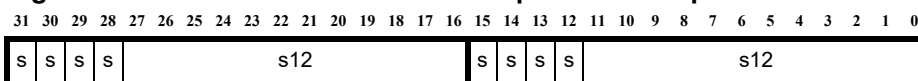


Figure 3-17 and Figure 3-18 illustrate the expansion of four 8-bits vectors (expanded u6 or s12 literals) to a word (32 bits).

Figure 3-17 Four 8-Bit Vectors with Expanded u6 Operands

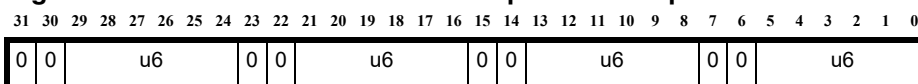
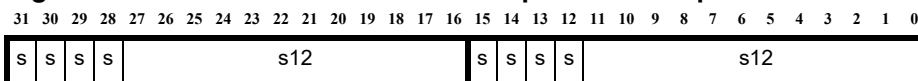


Figure 3-18 Four 16-Bit Vectors with Expanded s12 Operands



3.2.5 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 3-19 illustrates the expansion of a w6 in STD.

Figure 3-19 Expansion of a w6 in STD

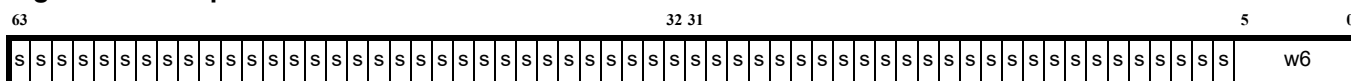
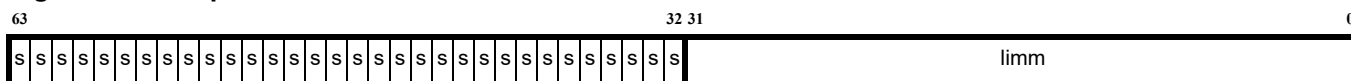


Figure 3-20 illustrates the expansion of a limm in STD

Figure 3-20 Expansion of a limm in STD



3.3 Floating-Point Data Formats

The floating-point unit supports the following data formats:

- “Half-Precision Floating-Point Data Formats” on page 114
- “Single-Precision Floating-Point Data Formats” on page 114
- “Double-Precision Floating-Point Data Formats” on page 115
- “NaN Formats” on page 115

3.3.1 Half-Precision Floating-Point Data Formats

When the hardware supports half-precision floating-point instructions (`-fpu_hp_option==true`), the format of the source operands and result are as specified by the IEEE-754 (2008) standard.

A half-precision floating-point number is represented in a core register as shown in [Figure 3-21](#).

Figure 3-21 Half-Precision Floating-Point Data

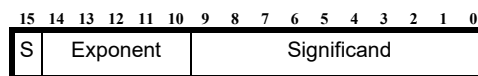


Table 3-1 Half-Precision Floating-Point Data Field Descriptions

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

3.3.2 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 3-22](#).

Figure 3-22 Single-Precision Floating-Point Data

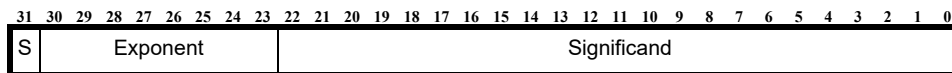


Table 3-2 Single-Precision Floating-Point Data Field Descriptions

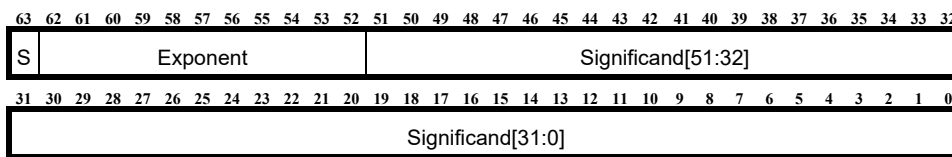
Field	Description
S	Sign part of the floating-point number 0—Positive number 1—Negative number
Exponent	Exponent part of the floating-point number

Table 3-2 Single-Precision Floating-Point Data Field Descriptions (Continued)

Field	Description
Significand	Significand part of the floating-point number

3.3.3 Double-Precision Floating-Point Data Formats

When hardware support for double-precision floating-point operations is configured (FPU_DP_OPTION == 1), a double-precision floating-point number is represented in core registers as shown in [Figure 3-23](#).

Figure 3-23 Double-Precision Floating-Point Data

The bit fields are described in the following table.

Table 3-3 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
Significand	Significand part of the floating-point number

3.3.4 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 single-precision floating-point unit handles the NaNs as follows:

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

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

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

**Note**

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

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

**Note**

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

$0 * \text{infinity}$

or

$\text{infinity} * 0$

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

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

**Attention**

For double-precision floating-point instructions enabled by FPU_DP_ASSIST, SNaN and QNaN are not strictly separated. Any NaN source operand is directly sent to output as the result. Illegal operation have NAN as the result, but not guaranteed as QNaN.

3.4 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.

3.4.1 64-Bit Data

[Figure 3-24](#) 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 3-25](#)

Figure 3-24 64-bit Register Data in Byte-Wide Memory, Little-Endian

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

Figure 3-25 64-bit Register Data in Byte-Wide Memory, Big-Endian

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

For information about the 64-bit data organization in a register pair, see [Figure 3-5](#) and [Figure 3-6](#).

3.4.2 32-Bit Data

All load or store, arithmetic, and logical operations support 32-bit data. [Figure 3-26](#) on page 118 show the data representation in a general purpose register. [Figure 3-27](#) on page 118 shows the little-endian representation in byte-wide memory. [Figure 3-28](#) on page 118 shows the big-endian representation.

Figure 3-26 Register containing 32-bit Data

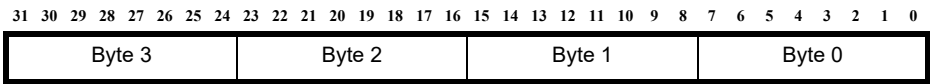


Figure 3-27 32-bit Register Data in Byte-Wide Memory, Little-Endian

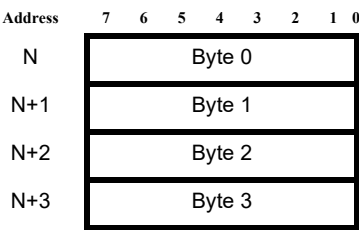
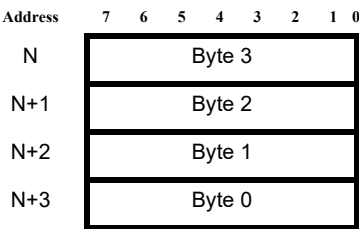


Figure 3-28 32-bit Register Data in Byte-Wide Memory, Big-Endian



3.4.3 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 3-29](#) on page 119 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 3-30](#) on page 119 shows the little-endian representation of 16-bit data in byte-wide memory. If the

ARCV2-based processor supports big-endian addressing, [Figure 3-31](#) on page 119 shows how the 16-bit data is stored in memory.

Figure 3-29 Register Containing 16-Bit Data

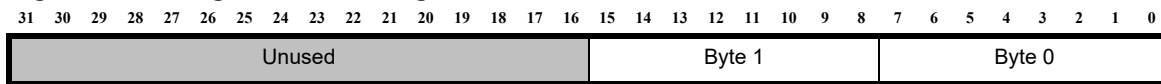


Figure 3-30 16-bit Register Data in Byte-Wide Memory, Little-Endian

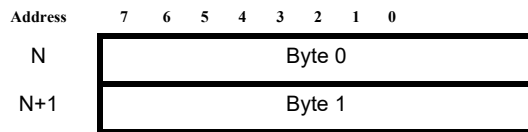


Figure 3-31 16-bit Register Data in Byte-Wide Memory, Big-Endian



3.4.4 8-Bit Data

The load and store operations support 8-bit data which can be promoted to a 32-bit value by using unsigned extend (EXTB) or signed extend (SEXB) instructions. [Figure 3-32](#) on page 119 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 3-33](#) on page 119 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 3-32 Register Containing 8-Bit Data

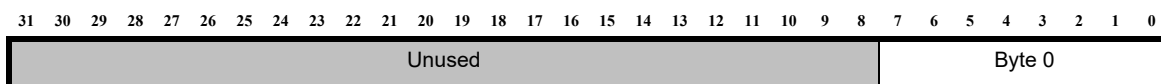
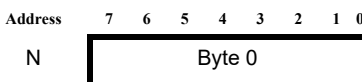
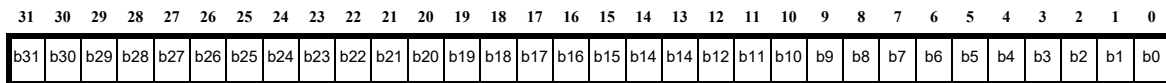


Figure 3-33 8-bit Register Data in Byte-Wide Memory



3.4.5 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 3-34 Register Containing 1-Bit Data

3.5 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.

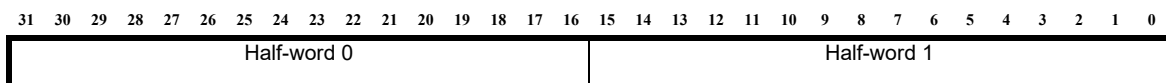
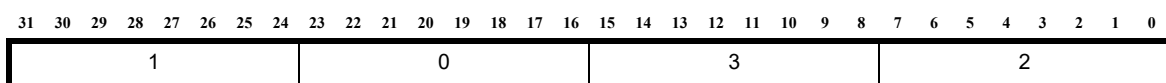
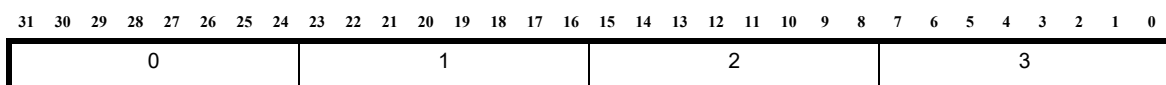
Figure 3-35 Half-word Numbering of a 32-Bit Instruction or Long-immediate Data Item

Figure 3-35 shows the number of two half-words in a 32-bit instruction or long-immediate data item.

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

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

Figure 3-36 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 3-37 shows the corresponding big-endian addresses.

Figure 3-36 Little-endian Offset of each Byte in a 32-Bit Instruction or Immediate**Figure 3-37 Big-endian Offset of each Byte in a 32-Bit Instruction or Immediate**

Similarly, Figure 3-38 illustrates the little-endian memory address offset of each byte within a 16-bit instruction, relative to the start address of that instruction. Figure 3-39 illustrates the big-endian memory offset of each byte in a 16-bit instruction.

3.6 Addressing Modes

Figure 3-38 Little-endian Memory Offset of each Byte in a 16-Bit Instruction

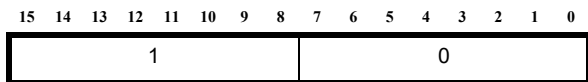
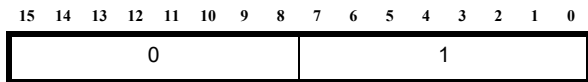


Figure 3-39 Big-endian Memory Offset of each Byte in a 16-bit Instruction



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

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

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

Table 3-4 Instruction Variants

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, half-word, or word)
<.x>	perform sign extension
<.di>	data cache bypass (load and store operations)
<.aa>	address writeback
<.T>	invert the default static branch prediction (see “Static Branch Prediction Mode” on page 322)

Table 3-5 Operand Description

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

3.7 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 3-1 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.

3.8 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.

3.9 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.

3.10 Compare and Branch Instruction

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

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

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

4

Core Architectural State Registers

4.1 Core Register Set

The ARCV2 programming model provides a set of general-purpose registers.

Figure 4-1 shows a summary of the core register set.

Figure 4-1 Core Register Map Summary

[illegible]

Figure 4-1 Core Register Map Summary (Continued)

r61	RESERVED	
r62	Long Immediate Data Indicator	
r63	Program Counter [PC_SIZE-1:2], Read-Only (PCL)	0 0

The default implementation of the core provides 32 general purpose 32-bit core registers. If required, up to 26 extension core registers can be defined, using APEX, when the core is configured. Alternatively, the core can be de-configured to provide only 16 general purpose 32-bit core registers. To reduce the physical size and power consumption of the core. When executing instructions encoded in 32-bit formats, the full range of core registers is available. [Table 4-1](#) on page 126 lists summarizes the accessibility of core registers from 16-bit and 32-bit instruction formats.

Table 4-1 Core Register Set

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

Table 4-1 Core Register Set (Continued)

Register	32-bit Instruction Function and Default Usage	16-bit Instruction Access to Register
r30	General purpose	Default access. Long immediate indicator for instructions with 'h' operand
r31 (BLINK) (see Link Registers, ILINK (r29), BLINK (r31))	Branch link register	JL_S, BL_S, J_S, PUSH_S, POP_S, MOV_S, CMP_S, ADD_S
r32 - r57r55	Extension core registers	
r56	Vector Stack Pointer	
r57	Vector Frame Pointer	
r58	Accumulator	
r59	Accumulator	
r60 (LP_COUNT) (see Loop Count Register, LP_COUNT (r60))	Loop counter	
r61	Reserved	Reserved
R62	Long immediate	MOV_S, CMP_S, ADD_S
R63 (PCL) (see Word-aligned Program Counter, PCL (r63))	Program counter (32-bit aligned)	

4.1.1 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 4-2](#) on page [127](#) lists the special register encoding and [Table 4-3](#) on page [128](#) lists the ABI usage support.

Table 4-2 16-bit Instruction Register Encoding

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

Table 4-2 16-bit Instruction Register Encoding (Continued)

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

4.1.2 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 4-3](#) on page 128. The RF_BUILD register is used to determine the configuration of core registers (see [Core Register File Configuration Register, RF_BUILD](#)).

Table 4-3 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	temp regs	•	•
r13-r15	saved regs	•	•
r16-r25	saved regs		
r26	GP (global pointer) (see Pointer Registers, GP (r26), FP (r27), SP (r28))		•
r27	FP (frame pointer) (see Pointer Registers, GP (r26), FP (r27), SP (r28))		•
r28	SP (stack pointer) (see Pointer Registers, GP (r26), FP (r27), SP (r28))		•
r29	ILINK (see Link Registers, ILINK (r29), BLINK (r31))		•
r30	General purpose register Note: This register is designated neither as a saved nor a temp register.		•
r31	BLINK (see Link Registers, ILINK (r29), BLINK (r31))		•

4.1.3 Multiple Register Banks

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

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 the number of configurable register banks is a power of two, the `STATUS32.RB` field can never point to an unimplemented register bank.

When the number of configurable register banks is not a power of two (3, 5, 6, or 7), the `STATUS32.RB` field value is constrained to never point to a register bank that does not exist. On an auxiliary write or other update of the `STATUS32` register `RB` field, if the update value is higher than the index of the last register bank, then the highest possible register bank index is written. This 'saturated' value is the value read back from the `STATUS32` register.

For example, in a configuration with three register banks (banks 0, 1 and 2) and a `STATUS32.RB` update attempts to write the value 3: the write value is modified to 2 and register bank 2 is selected.

The same mechanism is used when a debug host selects the register bank via the `DEBUG1.RB` field; any write of an index to a non-existent bank is saturated to the highest available bank.

Note that an attempt to switch to a register bank that is not available is a programming error.

[Table 4-4](#) specifies the set of replicated registers, for each value of `RGF_BANKED_REGS` when you are using multiple register file banks.



Note

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

Table 4-4 Multiple Bank Configuration

RGF_NUM_BANKS	RGF_NUM_REGS	Bank n
1 to 8	16	R0-R3, R10-R15, R26-R31
	32	R0-R31

4.1.3.1 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.

Second Bank SP and GP Initialization

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

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

Example:

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

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

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

4.1.3.2 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.



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

4.1.4 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 134.

4.1.5 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 4-3](#) on page 128.

4.1.6 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 consequently determines the size of the delay-slot instruction based solely on its major opcode. If the preceding branch or jump instruction sets the BLINK register (such as JL.D or BL.D), then the BLINK value defined by that instruction is the address of its delay-slot instruction plus the size of its delay-slot instruction, but always excluding any illegal LIMM operand.

4.1.7 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.

4.1.8 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 in your own assembly code because LP_COUNT maybe decremented if the program counter equals the loop end address. However, the MetaWare compiler can safely use this register as a general purpose register. For more information about the zero delay loop mechanism, see [LP<cc>](#) instruction details on [page 614](#).

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 4-1 Correct LP_COUNT Setup through a Register

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

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

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

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

4.1.9 Reserved Register (r61)

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

4.1.10 Immediate Data Indicator, L IMM (r62)

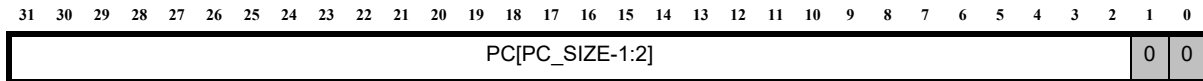
Register r62 is reserved for encoding long (32-bit) immediate (LIMM) data operands or to signify that the result of an instruction shall be discarded. Register r62 is therefore not available as a general purpose register.

When a L IMM 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 L IMM. In this case the L IMM forms part of the referencing instruction.

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

4.1.11 Word-aligned Program Counter, PCL (r63)

Figure 4-2 PCL Register



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

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

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

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

4.2 Error Protection on Core Registers

When the configuration option `-pipe_edc_option==true`, any single or double-bit errors in the 32 general-purpose registers are detected when these registers are read. When errors occur, the signal `edc_pipe_err` is asserted high.



Note

After reset, any single or double-bit errors in register r0 are detected even before any core register is read.

At reset, this register file is initialized along with EDC code. To avoid random ECC errors on the register file and X propagation into the control path of the core in simulation, all registers in the register file must be initialized by hardware at reset. At reset, EDC errors on the register r0 are detected.

4.3 Extension Core Registers

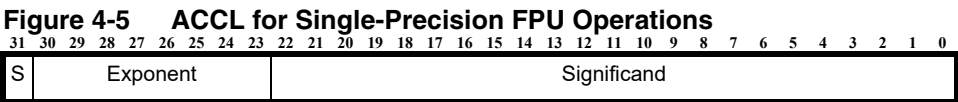
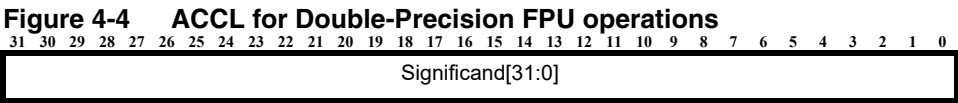
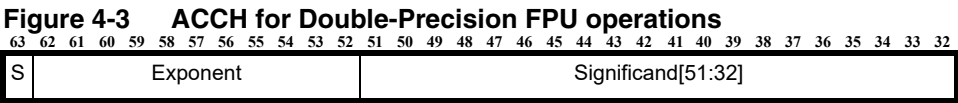
The register set is extensible in register positions 32-57 (r32-r57).

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

In this manual, the ACC register refers to the register pair comprising the accumulator (ACCH, ACCL), according to the definition of a register pair given in [“64-Bit Data”](#) on page 117.

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



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 writeback. In this case the writeback value is discarded.

4.3.1 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 131.

4.4 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.

4.4.1 Baseline Auxiliary Registers

[Table 4-5](#) lists the baseline set of auxiliary registers. The access code definitions of these registers are explained in [Table 4-6](#).

Table 4-5 Baseline Auxiliary Register Set

Number	Auxiliary Register Name	Description
Current Architectural State		

Table 4-5 Baseline Auxiliary Register Set (Continued)

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

4.4.2 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 4-6](#). 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.

**Note**

Access to auxiliary registers through side-effect, such as altering flags in the STATUS32 register, is governed by the privilege requirements of any instruction that makes an implicit read or write to an auxiliary register. [Table 4-6](#) 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 4-6](#), “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 4-6 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	•		•		•	-
R	P		•		•	-
w		•		•	-	•
rw	•	•	•	•	•	•
W		P		•	-	•
rW	•	P	•	•	•	•
RW	P	P	•	•	•	•
rG	•		•		•	•
RG	P		•		•	•

4.4.3 Optional Instruction Set Auxiliary Registers

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

Table 4-7 Zero Overhead Loop Auxiliary Registers (`lpc_size > 0`)

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

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

Table 4-8 Indexed Table Auxiliary Register

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

4.4.4 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.

Table 4-9 XPU, User Extension Permission Auxiliary Register

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

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

Table 4-10 Extension Flags Register, XFLAGS

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

4.4.5 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.

4.4.6 Reset Value of Auxiliary Registers

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

4.4.7 Auxiliary Registers Holding Address Values

A significant number of the proprietary auxiliary registers contain address values. When an ARCV2 based processor is configured with a memory management unit (MMU) those addresses can be virtual or physical. The ARCV2 architecture specifies that all address values in auxiliary registers are to be considered virtual unless they are explicitly defined as containing physical addresses.

All of the following auxiliary registers hold physical addresses when an MMU is configured:

- Exception Fault Address: [“EFA”](#) on page 170
- MMU API registers: [“TLB Page Descriptor Register for 32-Bit Physical Address, TLBPD1”](#) and [“TLB Page Descriptor Register for 64-Bit Physical Address, TLBPD1_HI”](#)
- Instruction cache registers: [“IC_IVIR”](#) on page 934 and [“IC_ENDR”](#) on page 936
- Data cache registers: [“DC_STARTR”](#) on page 970 and [“DC_ENDR”](#) on page 971
- All MPU base addresses (an MPU can be configured only in systems without MMU and are therefore implicitly physical)
- I/O coherency and SLC cache-ability aperture base address register: [BIU_IO_0_COH_AP0_BASE](#)
- All SLC internal auxiliary registers
- All ARConnect auxiliary registers

4.4.8 Loop Start Register, LP_START

Address: 0x02

Access: rw

Reset: 0x0000_0000

Figure 4-6 LP_START Register (PC_SIZE == 32)

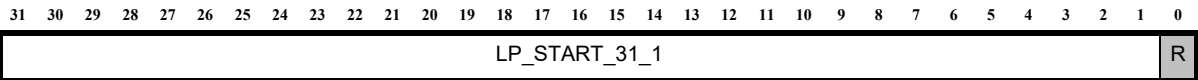
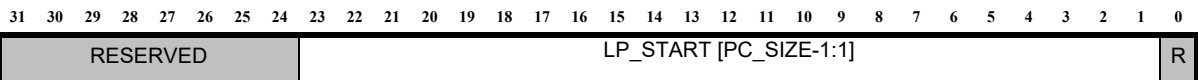


Figure 4-7 LP_START Register (PC_SIZE < 32)



The loop start (LP_START) register contains the address at which the current zero delay loop begins. [Figure 4-6](#) on [page 139](#) shows the format of this register. The loop start register can be set up with the LPcc instruction or can be manipulated using the auxiliary register access instructions (see [LR](#) and [SR](#)) format, and therefore, its size is determined by the PC_SIZE configuration parameter.

Programming cautions may exist when using the loop control registers. For more information, see [LP<cc>](#) instruction on [page 614](#).

Table 4-11 LP_START Field Description

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

4.4.9 Loop End Register, LP_END

Address: 0x03

Access: rw

Reset: 0x0000_0000

Figure 4-8 LP_END Register

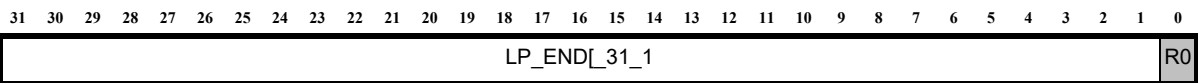
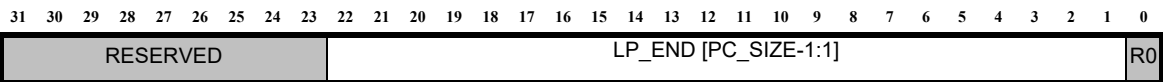


Figure 4-9 LP_END Register (PC_SIZE < 32)



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

 **Note**

Care should be taken to align LP_END with an instruction boundary position. An Illegal Instruction Sequence exception is raised when LP_END is at 16-bit offset position immediately after an RTIE (32-bit) instruction.

Programming cautions may exist when using the loop control registers. For more information, see the LP<cc> instruction on [page 614](#) and “[Loop Count Register, LP_COUNT \(r60\)](#)” on [page 132](#).

Table 4-12 LP_END Field Description

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

4.4.10 Core Identity Register, IDENTITY

Address: 0x04

Access: r

Figure 4-10 IDENTITY Register

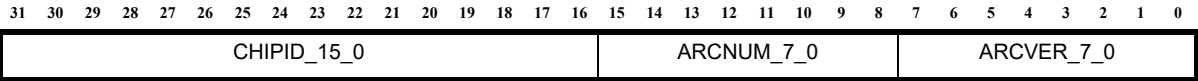


Figure 4-10 on page 141 shows the identity register (IDENTITY). This register contains the following fields:

Table 4-13 IDENTITY Field Descriptions

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

Table 4-13 IDENTITY Field Descriptions (Continued)

Field	Bit	Description
ARCOVER_7_0	[7:0]	ARC baseline instruction set version number All non-enumerated values are reserved. ■ 0x41: ARC EM processor version 1.1a ■ 0x42: ARC EM processor version 3.0; added support for micro-DMA controller, CryptoPack extension, PDM interface, NV_ICCM, wide ICCM and DCCM, unaligned load and store. See the Release Notes for more information. ■ 0x43: ARC EM processor version 4.0 and ARC SEM version 1.0; Support for the address and data bus scrambling for caches, secure kernel and user modes, side-channel protection, safety island, secure debug, micro-DMA enhancements, and so on. See the Release Notes for details. ■ 0x44: ARC EM processor version 5.10a and ARC SEM version 2.10a that includes support for the non-maskable imprecise exception and updated EFA behavior (unlike previous releases, EFA now always exists in the processor). ■ 0x45: ARC EM Processor version 5.50a that includes support for the TASK_SWITCH component and Overlay Manager. ■ 0x46: ARC EM Processor version 5.60a that supports the Data access from absent memory exception. ■ 0x50: ARC HS processor version 1.0; first release of the ARC HS processor. ■ 0x51: ARC HS version 2.0; added multi-core support, added support for L2 cache, MMU support for multiple page sizes, increased the size of the supported CCMs, added support for up to eight register banks, added support for performance counters, ■ 0x52: ARC HS version 2.1; added single-cycle option for DCCM and data cache, dropped support for BVCI ■ 0x53: ARC HS version 3.0 and Embedded Vision processor (EV6x); support for cluster shared memory, Vector DSP ISA, APEX enhancements to improve performance and usability, added the DBNZ instruction, added PDM (Power Domain Management) unit to support external PMU, added FastMath extension pack as an option. See the Release Notes for details. ■ 0x54: ARC HS version 3.10a, 3.50a, 3.60a, and 4.00a, ARC VPX5 1.00a, EV6x (2.20a) and EV7x (3.00a) processors: support for shared memory configuration of CSM and L2 cache, unified DMI port, support for two peripheral ports. ■ 0x60 to 0xFF = Reserved

4.4.11 Program Counter, PC

Address: 0x06

Access: rG

Reset: 0x0000_0000

Figure 4-11 PC Register Addressing Full 32-bit Address Space

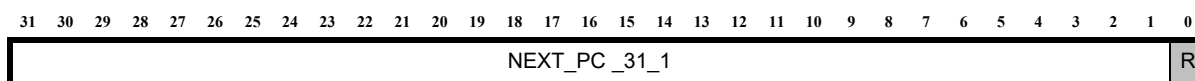
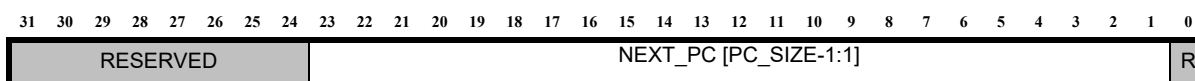


Figure 4-12 PC Register Addressing a Reduced Address Space (PC_SIZE == 32)



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.

When the configuration option `-pipe_edc_option==true`, any single or double-bit errors in this register are detected, and the signal `edc_pipe_err` is asserted high. At reset, this register is initialized along with EDC code.

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 4-14](#) lists the addressable range for different PC width configurations.

Table 4-14 ARCV2 Addressable Range vs. PC Size

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

Table 4-15 PC Field Description

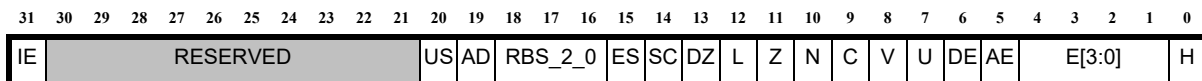
Field	Bit	Reset	Description
NEXT_PC__31_1_	[31:1]	0	Address (MSB 31 bits) of the next instruction to be committed
R0	[0]	0	Reserved. read-as-zero; ignored on write

4.4.12 Status Register, STATUS32

Address: 0x0A

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

Figure 4-13 STATUS32 Register



The status register, STATUS32 performs the following functions:

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

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

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

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

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

STATUS32 Behavior on Interrupt Entry or Exit

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

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

Table 4-16 STATUS32 Bit-Field Definitions, Read / Write Accessibility , and Reset Value

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

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.

US

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

When US==0:

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

When US==1:

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

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

AD

The AD bit is present only in processors that support non-aligned data references. The AD bit indicates whether alignment checks on data memory references are disabled. If this bit is set to 1, all data memory alignment checks, except for the EX, ENTER_S, and LEAVE_S instructions, are disabled, and the processor does not raise EV_Misaligned ([Misaligned Data Access](#)) exceptions. This information is accessible to software by inspecting the NON_ALIGNED field in the [Instruction Set Configuration Register, ISA_CONFIG](#) register. If this feature is not supported, or if the AD bit is set to 0, natural alignment of all data references, as defined in “[Data Layout in Memory](#)” on page 116 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.

RB_2_0_

The RB field indicates the register bank that the processor is currently using. In kernel mode, use the KFLAG instruction to write to the STATUS32.RB[2:0] to select a register bank. Accesses to core registers from thereon are made to the registers in the selected register bank. The configuration parameter,

`RGF_NUM_BANKS`, defines the number of banks in a processor. The value written to the `STATUS32.RB` field must be less than `RGF_NUM_BANKS`.

**Note**

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

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

When the number of configurable register banks is a power of two, the `STATUS32.RB` field can never point to an unimplemented register bank.

When the number of configurable register banks is not a power of two (3, 5, 6, or 7), the `STATUS32.RB` field value is constrained to never point to a register bank that does not exist. On an auxiliary write or other update of the `STATUS32` register `RB` field, if the update value is higher than the index of the last register bank, then the highest possible register bank index is written. This 'saturated' value is the value read back from the `STATUS32` register.

For example, in a configuration with three register banks (banks 0, 1 and 2) and a `STATUS32.RB` update attempts to write the value 3: the write value is modified to 2 and register bank 2 is selected.

Note that an attempt to switch to a register bank that is not available is a programming error.

ES

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

SC

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

- [User Stack Region Top Address, `USTACK\_TOP`](#) and [User Stack Region Base Address, `USTACK\_BASE`](#) (`KSTACK_TOP`, `USTACK_TOP`), (`USTACK_BASE`, `KSTACK_BASE`)
- [Stack Region Configuration Register, `STACK\_REGION\_BUILD`](#) (`STACK_REGION_BUILD` BCR)

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

DZ

`DZ` indicates whether the `EV_DivZero` exception is enabled on division by zero. When `DZ` is set to 1, any attempt to execute a `DIV`, `DIVU`, `REM`, or `REMU` instruction, with a divisor of 0, raises an `EV_DivZero` exception. When `DZ` is set to 0, a division by zero does not raise an exception. The `DZ` bit is ignored on write.

and read as zero when there is no hardware divider configured, regardless of operating mode. The DZ bit is cleared on interrupt or exception entry, and is restored on return from an interrupt or exception.

L

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

U

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

DE

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

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

AE

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

E[3:0]

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

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

**Note**

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

H

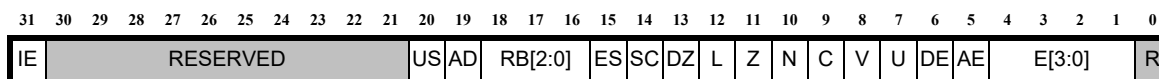
The H bit when set to 1 indicates that the core is halted. All fields, except the H bit, are set to 0 when the processor is [Reset](#). The H bit is set depending on the configuration of the processor run state on [Reset](#).

4.4.13 Status Register Priority 0, STATUS32_P0

Address: 0x0B

Access: RW

Figure 4-14 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.

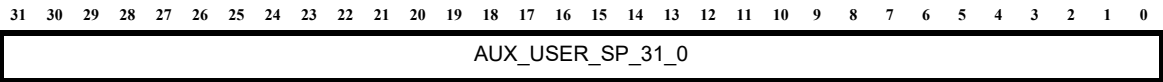
4.4.14 Saved User Stack Pointer, AUX_USER_SP

Address: 0x0D

Access: RW

Reset: 0x0000_0000

Figure 4-15 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. This instruction can also be used to efficiently exchange LP_START or LP_END with a core register during full context save and restore sequences.

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

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

Table 4-17 AUX_USER_SP Field Description

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

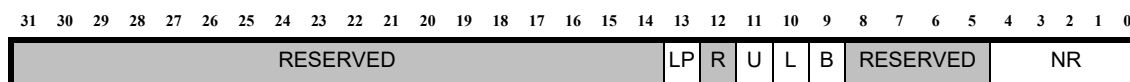
4.4.15 Interrupt Context Saving Control Register, AUX_IRQ_CTRL

Address: 0x0E

Access: RW

Reset: 0x0000_0000

Figure 4-16 AUX_IRQ_CTRL Register



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

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

On reset, this register contains 0x00000000.

Table 4-18 AUX_IRQ_CTRL Field Description

Field	Bit	Reset	Description
NR	[4:0]	0	Indicates number of general-purpose register pairs saved, from 0 to 8/16. This register saturates at 16 when the 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 Table 4-3 .
RESERVED	[8:5]	0	Reserved. read-as-zero; ignored on write
B	[9]	0	Indicates whether to save and restore BLINK (ignored if NR is greater than or equal to 16.)
L	[10]	0	Indicates whether to save and restore loop registers (LP_COUNT, LP_START, LP_END)
U	[11]	0	Indicates if user context is saved to user stack
RESERVED	[12]	0	Reserved. read-as-zero; ignored on write
LP	[13]	0	Indicates whether to save and restore code-density registers (EI_BASE, JLI_BASE, LDI_BASE)
RESERVED	[31:14]	0	Reserved. read-as-zero; ignored on write

4.4.16 Interrupt Vector Base Register, INT_VECTOR_BASE

Address: 0x25

Access: RW

Reset: Based on the ARChitect configuration

Figure 4-17 INT_VECTOR_BASE Register (PC_SIZE == 32)

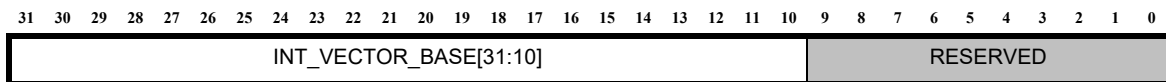
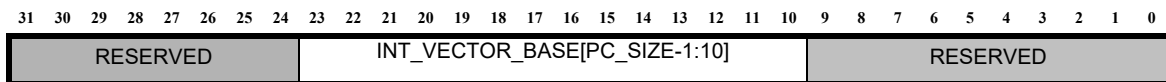


Figure 4-18 INT_VECTOR_BASE Register (PC_SIZE < 32)



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 187). 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 HS family of processors, this address is within the upper 2 GBytes of the memory space. This design avoids double-fault exceptions from occurring when accessing the vector table to service a first exception. Fetching the vectors is then not subject to TLB misses or other TLB exceptions.

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

Table 4-19 INT_VECTOR_BASE Field Description

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

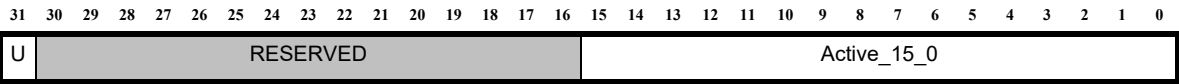
4.4.17 Active Interrupts Register, AUX_IRQ_ACT

Address: 0x43

Access: RW

Reset: 0x0000_0000

Figure 4-19 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 212.

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.

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

Table 4-20 AUX_IRQ_ACT Field Description

Field	Bit	Reset	Description
Active_15_0	[15:0]	0	Active Interrupt Priority level
Reserved	[30:16]	0	Reserved. Read-as-zero; ignored on write.
U	[31]	0	Snapshot of the STATUS32.U bit when an interrupt is taken at a point where Active[15:0] == 0.

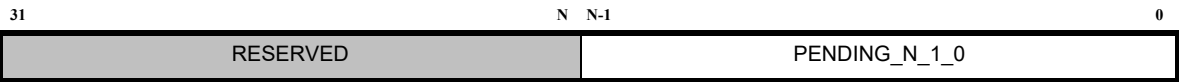
4.4.18 Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING

Address: 0x200

Access: R

Reset: 0x0000_0000

Figure 4-20 IRQ_PRIORITY_PENDING Register



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

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.

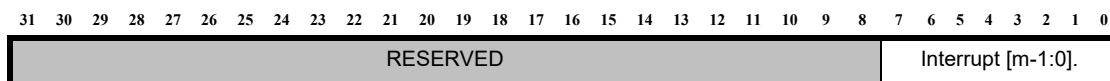
Table 4-21 IRQ_PRIORITY_PENDING Field Description

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

4.4.19 Software Interrupt Trigger, AUX_IRQ_HINT

Address: 0x201
 Access: RW
 Reset: 0x0000_0000

Figure 4-21 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).

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

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



Note

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

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

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

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

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

4.4.20 Interrupt Priority Register, IRQ_PRIORITY

Address: 0x206

Access: RW

Reset: 0x0000_0000

Figure 4-22 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. You can read and write to the IRQ_PRIORITY register in kernel mode; this register is protected and inaccessible in user mode.

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

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

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

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

Table 4-22 IRQ_PRIORITY Field Description

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

Table 4-22 **IRQ_PRIORITY Field Description**

Field	Bit	Reset	Description
Reserved	[31:n]	0	Reserved. Read-as-zero; ignored on write.

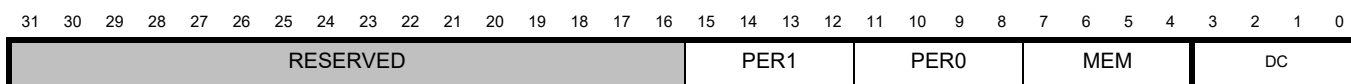
4.4.21 Data Memory Attribute Register, DATA_MEM_ATTR

Address: 0x20B; This register exists in the cluster auxiliary space and is shared by all the cores.

Access: RW

Reset 0x0000_0003

Figure 4-23 DATA_MEM_ATTR



The memory attributes (for example: bufferable, cacheable) of a bus transaction is programmable through this register. The fields in this register comply with the AxCACHE AXI4 specification. This register exists in the cluster auxiliary space and is shared by all the cores.



Note

A best practice is to execute the SYNC instruction before writing to this register. This ensures that all outstanding memory transactions (with the older attributes) are completed before programming the new attributes.

Table 4-23 DATA_MEM_ATTR Field Descriptions

Field	Bit	Reset	Description
DC	[3:0]	3	Attributes of data cache refill and copy back transactions ■ bit [0]: bufferable - 0: non bufferable - 1: bufferable ■ bit [1]: modifiable (cacheable) - 0: non cacheable - 1: cacheable ■ bit [2]: other allocate (read allocate) - 0: no read allocate - 1: read allocate ■ Bit [3]: allocate (write allocate) - 0: no write allocate - 1: write allocate These bits are available only if data cache is configured in your processor.

Table 4-23 DATA_MEM_ATTR Field Descriptions (Continued)

Field	Bit	Reset	Description
MEM	[7:4]	0	Attributes of uncached memory read and write transactions ■ bit [4]: bufferable - 0: non bufferable - 1: bufferable ■ bits [7:5]: reserved (RAZ/IOW) These bits are available only if uncached memory is configured.
PER0	[11:8]	0	Attributes of peripheral 0 read and write transactions ■ bit [8]: bufferable - 0: non bufferable - 1: bufferable ■ bits [11:9]: reserved (RAZ/IOW) These bits are available only if peripheral 0 interface is configured.
PER1	[15:12]	0	Attributes of peripheral 1 read and write transactions ■ bit [12]: bufferable - 0: non bufferable - 1: bufferable ■ bits [15:13]: reserved (RAZ/IOW) These bits are available only if peripheral 1 interface is configured.
RESERVED	[31:16]	0	Reserved. Read-as-zero; ignored on write.

4.4.22 Jump and Link Indexed Base Address, JLI_BASE

Address: 0x290

Access: rw

Reset 0x0000_0000

Figure 4-24 JLI_BASE Register when PC_SIZE == 32

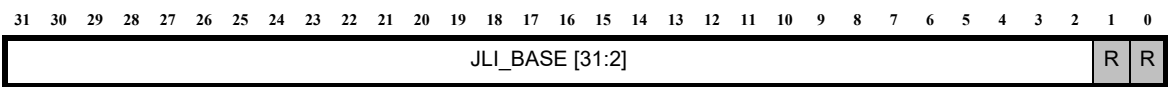
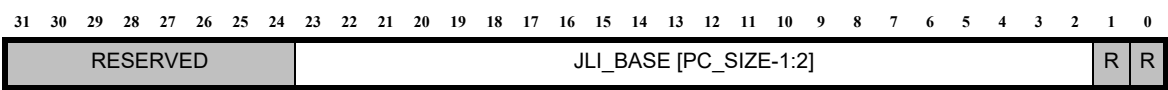


Figure 4-25 JLI_BASE Register when PC_SIZE < 32



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.

4.4.23 Load Indexed Base Address, LDI_BASE

Address: 0x291

Access: rw

Reset 0x0000_0000

Figure 4-26 LDI_BASE Register when ADDR_SIZE == 32

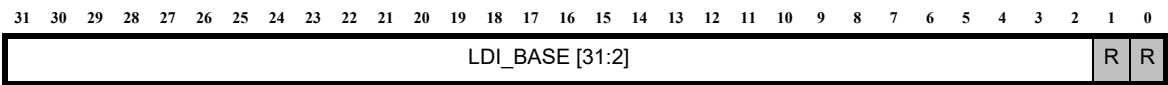
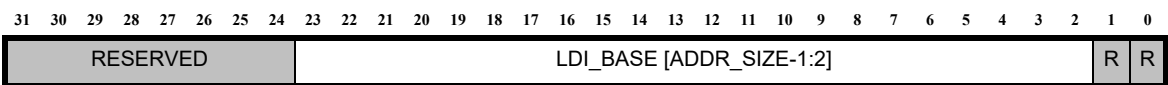


Figure 4-27 LDI_BASE Register when ADDR_SIZE < 32



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.

4.4.24 **Execute Indexed Base Address, EI_BASE**

Address: 0x292

Access: rw

Reset 0x0000_0000

Figure 4-28 EI_BASE Register when PC_SIZE == 32

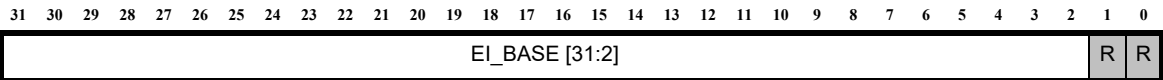
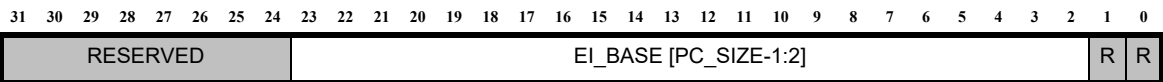


Figure 4-29 EI_BASE Register when PC_SIZE < 32



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.

4.4.25 Exception Return Address, ERET

Address: 0x400

Access: RW

Reset 0x0000_0000

Figure 4-30 ERET Register when PC_SIZE == 32

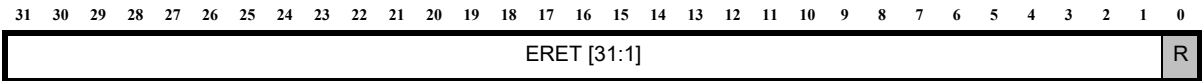
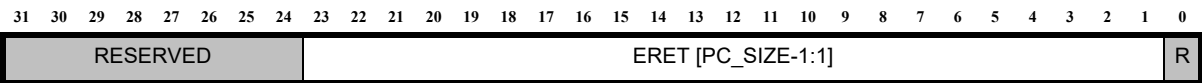


Figure 4-31 ERET Register when PC_SIZE < 32



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

For a list of ERET values for each exception types, see [Table 5-5](#).

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.

4.4.26 Exception Return Branch Target Address, ERBTA

Address: 0x401

Access: RW

Reset 0x0000_0000

Figure 4-32 ERBTA Register when PC_SIZE == 32

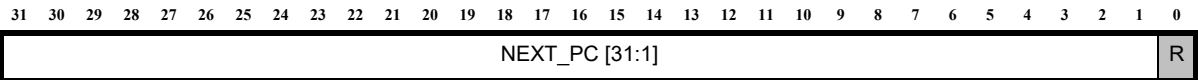
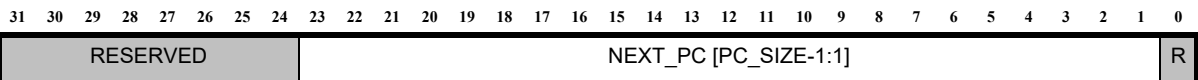


Figure 4-33 ERBTA Register when PC_SIZE < 32



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.

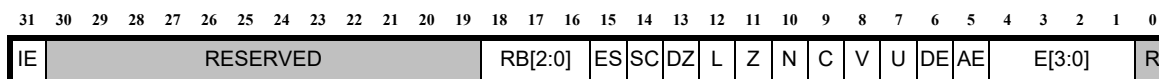
4.4.27 Exception Return Status, ERSTATUS

Address: 0x402

Access: RW

Reset 0x0000_0000

Figure 4-34 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.

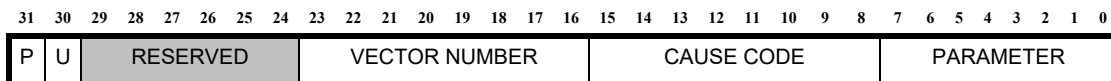
4.4.28 Exception Cause Register, ECR

Address: 0x403

Access: RW

Reset 0x0000_0000

Figure 4-35 ECR Register



The Exception Cause register (ECR) provides information about the source of the most recent exception to have been taken. [Figure 4-35](#) on page 168 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 5-3](#) on page 228 for a list of exception vectors and their assigned vector numbers.

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

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

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

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

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



Note

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

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

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

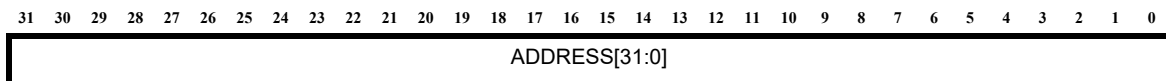
Table 4-24 ECR Field Description

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

4.4.29 Exception Fault Address, EFA

Address: 0x404
 Access: RW
 Reset 0x0000_0000

Figure 4-36 EFA Register



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

Whenever an exception is raised, the EFA register generally records a fault address based on the exception type. For information about what value the EFA register holds for each exception, see [Table 5-5](#) on page 238.

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

When 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.

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 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 includes the faulty bits, 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.

Table 4-25 EFA Field Description

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

4.4.30 Interrupt Cause Registers, ICAUSE

Address: 0x40A

Access: R

Reset 0x0000_0000

Figure 4-37 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. The ICAUSE register must be read after two instructions for a single issue and four instructions for a dual issue in the interrupt service routine to get the updated value.

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.

Table 4-26 ICAUSE Field Description

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

4.4.31 Interrupt Select, IRQ_SELECT

Address: 0x40B

Access: RW

Reset 0x0000_0000

Figure 4-38 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 4-27 IRQ_SELECT Field Description

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

4.4.32 Interrupt Enable Register, IRQ_ENABLE

Address: 0x40C

Access: RW

Reset 0x0000_0000

Figure 4-39 IRQ_ENABLE Register

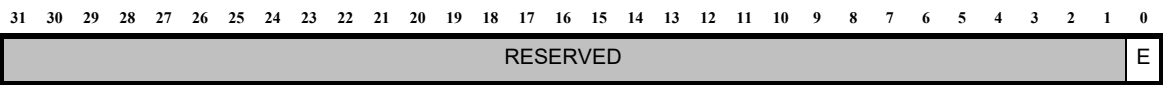


Table 4-28 IRQ_ENABLE Field Description

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

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

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.

4.4.33 Interrupt Trigger Register, IRQ_TRIGGER

Address: 0x40D

Access: RW

Reset 0x0000_0000

Figure 4-40 IRQ_TRIGGER Register

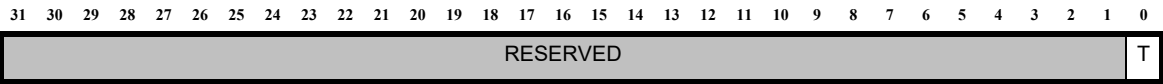


Table 4-29 IRQ_TRIGGER Field Description

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

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

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

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

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

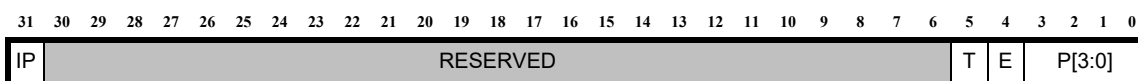
4.4.34 Interrupt Status Register, IRQ_STATUS

Address: 0x40F

Access: R

Reset 0x0000_0000

Figure 4-41 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.

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 4-30 IRQ_STATUS Field Description

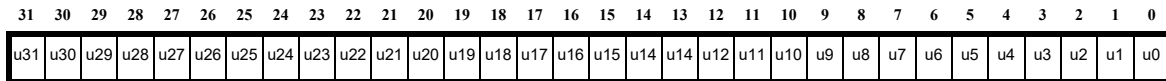
Field	Bit	Reset	Description
P[3:0]	[3:0]	0	Value of Interrupt Priority Register, IRQ_PRIORITY , for the selected interrupt
E	[4]	0	Value of Interrupt Enable Register, IRQ_ENABLE , for the selected interrupt
T	[5]	0	Value of Interrupt Trigger Register, IRQ_TRIGGER , for the selected interrupt
RESERVED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

4.4.35 User Mode Extension Enable Register, XPU

Address: 0x410

Access: RW

Figure 4-42 XPU Register



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

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.

4.4.36 Branch Target Address, BTA

Address: 0x412

Access: RW

Reset 0x0000_0000

Figure 4-43 BTA Register (PC_SIZE == 32)

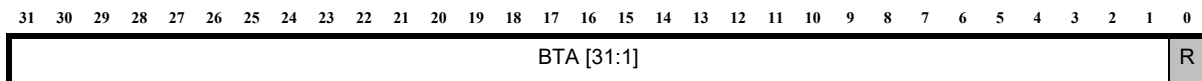
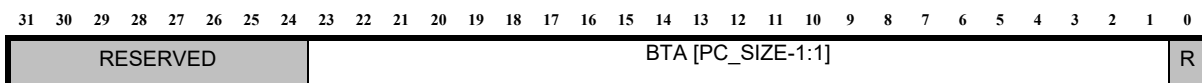


Figure 4-44 BTA Register (PC_SIZE < 32)



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.

4.4.37 Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL

Address: 0x415

Access: W

Reset 0x0000_0000

Figure 4-45 IRQ_PULSE_CANCEL Register

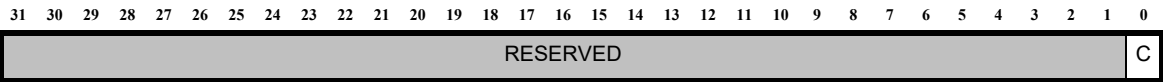


Table 4-31 IRQ_PULSE_CANCEL Field Description

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

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

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

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

4.4.38 Interrupt Pending Register, IRQ_PENDING

Address: 0x416

Access: R

Reset 0x0000_0000

Figure 4-46 IRQ_PENDING Register

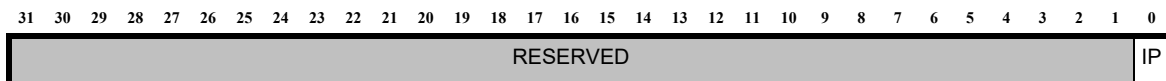


Table 4-32 IRQ_PENDING Field Description

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

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

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

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

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

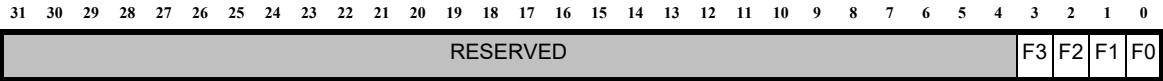
On reset, this register contains 0x00000000 for configured interrupts; if Interrupt [m-1:0] is greater than M+16-1 or less than 16, this register is read as zero. Where, M is the number of interrupts configured, $m = \text{ceil}(\log_2(M + 16))$, and ceil is the function to round to the next higher integer.

4.4.39 User Extension Flags Register, XFLAGS

Address: 0x44F

Access: RW

Figure 4-47 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.

4.5 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 4-33](#) summarizes the build configuration registers for components that are described in this document.

Table 4-33 Build Configuration Registers

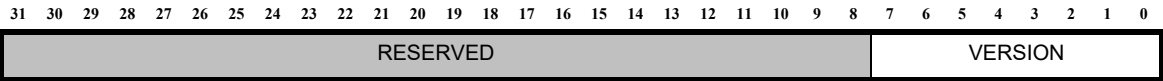
Number	Name	Access
0x60	Build Configuration Registers Version, BCR_VER	R
0x63	BTA Configuration Register, BTA_LINK_BUILD	R
0x68	Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD	R
0x6E	Core Register File Configuration Register, RF_BUILD	R
0x7B	Multiplier Configuration Register, MULTIPLY_BUILD	R
0x7C	Swap Instruction Configuration Register, SWAP_BUILD	R
0x7D	Normalize Instruction Configuration Register, NORM_BUILD	R
0x7E	Min/Max Instruction Configuration Register, MINMAX_BUILD	R
0x7F	Barrel Shifter Configuration Register, BARREL_BUILD	R
0xC1	Instruction Set Configuration Register, ISA_CONFIG	R
0xF3	Interrupt Build Configuration Register, IRQ_BUILD	R
0xF8	ARCV2 ISA Profile Register, ISA_PROFILE	R
0xF9	Micro Architecture Configuration Register, MICRO_ARCH_BUILD	R

4.5.1 Build Configuration Registers Version, BCR_VER

Address: 0x60

Access: R

Figure 4-48 BCR_VER Register



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

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

Table 4-34 BCR_VER Field Descriptions

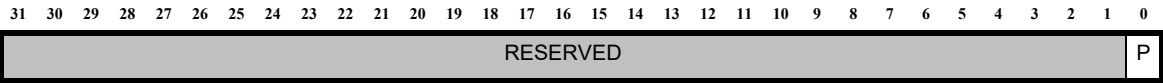
Field	Bit	Description
VERSION	[7:0]	Version of Build Configuration Registers. All non-enumerated values are reserved. 0x1: Indicates that the BCR Region is at addresses 0x60-0x7F and ISA_CONFIG only 0x2: Indicates that the BCR Region is at addresses 0x60-0x7F and 0xC0-0xFF 0x3: Indicates that the BCR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

4.5.2 BTA Configuration Register, BTA_LINK_BUILD

Address: 0x63

Access: R

Figure 4-49 BTA_LINK_BUILD Register



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

Table 4-35 BTA_LINK_BUILD Field Descriptions

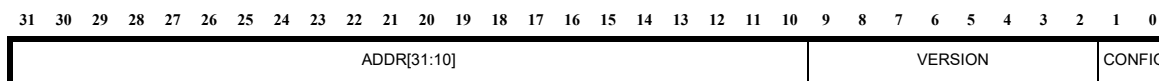
Field	Bit	Description
P	[0]	Presence of BTA Registers. All non-enumerated values are reserved. 0x0 = BTA_L1 and BTA_L2 registers are absent
RESERVED	[31:1]	Reserved. Read-as-zero; ignored on write.

4.5.3 Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD

Address: 0x68

Access: R

Figure 4-50 VECBASE_AC_BUILD Register



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

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

Table 4-36 VECBASE_AC_BUILD Field Descriptions

Field	Bit	Description
CONFIG	[1:0]	Configuration of the reset value for INT_VECTOR_BASE. All non-enumerated values are reserved. 0x0: Address is supplied with the <code>-intvbase_preset</code> option, when <code>-intvbase_ext</code> is false. 0x1: Address is supplied through the <code>intvbase_in</code> core input signal, when the <code>-intvbase_ext</code> option is true. See Table 4-37. Other values are reserved.
VERSION	[9:2]	Version of Interrupt Unit. All non-enumerated values are reserved. 0x04 = ARCV2 Interrupt Unit
ADDR[31:10]	[31:10]	Interrupt vector base address, excluding the lower 10 bits, that is configured when the core comes out of reset.

Table 4-37 Reset Value of the Interrupt Vector Base

<code>-intvbase_ext==true</code>	<code>-mcip_has_intvbase_control==true</code>	What drives the Reset Value
Yes	No	<code>intvbase_in[21:0]</code> . This signal is connected to the core directly.

Table 4-37 Reset Value of the Interrupt Vector Base

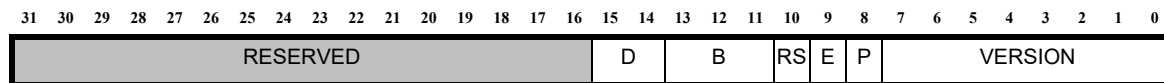
<code>-intvbase_ext==true</code>	<code>-mcip_has_intvbase_control==true</code>	What drives the Reset Value
Yes	Yes	You can use either of the following: ■ Programmable using the ARConnect IVC. ■ The <code>intvbase_in[21:0]</code> signal is connected to ARConnect. In turn, ARConnect routes the value on this signal to the cores.
No	Yes	At least one core in a multi-core configuration must have <code>-intvbase_ext==true</code>. Otherwise, the reset value of the interrupt vector base of a core is determined by its <code>-intvbase_preset</code> configuration option.
No	No	The reset value of the interrupt vector base of a core is determined by its <code>-intvbase_preset</code> configuration option.

4.5.4 Core Register File Configuration Register, RF_BUILD

Address: 0x6E

Access: R

Figure 4-51 RF_BUILD Register



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

Table 4-38 RF_BUILD Field Descriptions

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

Table 4-38 RF_BUILD Field Descriptions (Continued)

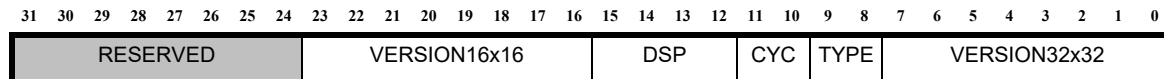
Field	Bit	Description
D	[15:14]	Number of duplicated registers in each additional register bank ■ 0x0: 4 duplicated registers ■ 0x1: 8 duplicated registers ■ 0x2: 16 duplicated registers ■ 0x3: 32 duplicated registers Note: This field returns 0 when RF_BUILD.B == 0, that is there are no additional register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

4.5.5 Multiplier Configuration Register, MULTIPLY_BUILD

Address: 0x7B

Access: R

Figure 4-52 MULTIPLY_BUILD Register



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

Table 4-39 MULTIPLY_BUILD Field Descriptions

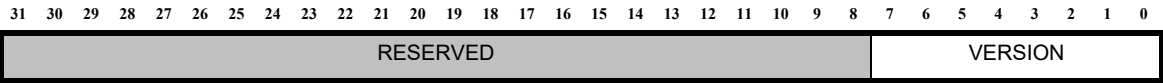
Field	Bit	Description
VERSION32x32	[7:0]	Version of 32x32 Multiply. All non-enumerated values are reserved. 0x6: ARCV2 32x32 Multiply
TYPE[1:0]	[9:8]	Type of Multiply. All non-enumerated values are reserved. 0x0: Serial multiplier, computing 4-bits of the result per cycle 0x1: Parallel multiplier, with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

4.5.6 Swap Instruction Configuration Register, SWAP_BUILD

Address: 0x7C

Access: R

Figure 4-53 SWAP_BUILD Register



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

Table 4-40 SWAP_BUILD Field Descriptions

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

4.5.7 **Normalize Instruction Configuration Register, NORM_BUILD**

Address: 0x7D

Access: R

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

Figure 4-54 NORM_BUILD Register

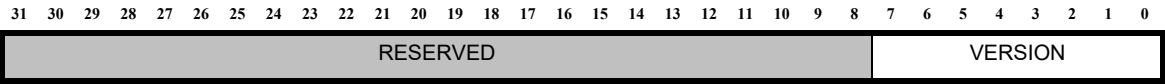


Table 4-41 NORM_BUILD Field Descriptions

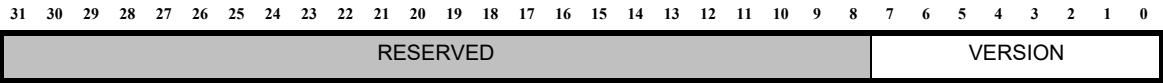
Field	Bit	Description
VERSION	[7:0]	Version of Norm. All non-enumerated values are reserved. ■ 0x1 : Reserved ■ 0x2 : ARCompact (ARCV1) version ■ 0x3 : ARCV2 version
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

4.5.8 Min/Max Instruction Configuration Register, MINMAX_BUILD

Address: 0x7E

Access: R

Figure 4-55 MINMAX_BUILD Register



The MIN/MAX instruction configuration register, MINMAX_BUILD, contains the version of the MIN and MAX instructions.

Table 4-42 MINMAX_BUILD field descriptions

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

4.5.9 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 4-56 BARREL_BUILD Register

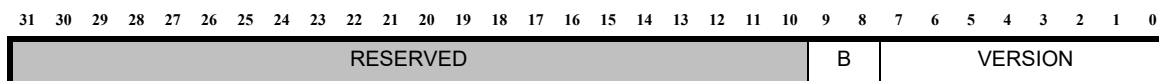


Table 4-43 BARREL_BUILD Field Descriptions

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

4.5.10 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 99.

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

The values of each field in ISA_CONFIG select the configuration of the relevant option in accordance with the interpretation defined by the section on “Configurability” on page 99, and Table 2-2.

Figure 4-57 ISA_CONFIG Build Register

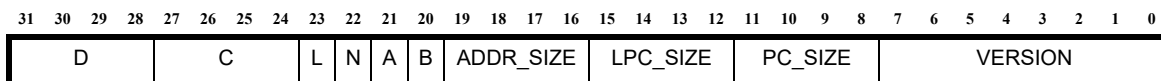


Table 4-44 ISA_CONFIG Field Descriptions

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

Table 4-44 ISA_CONFIG Field Descriptions (Continued)

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

Table 4-44 ISA_CONFIG Field Descriptions (Continued)

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

4.5.11 Interrupt Build Configuration Register, IRQ_BUILD

Address: 0xF3

Access: R

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

Figure 4-58 IRQ_BUILD Register

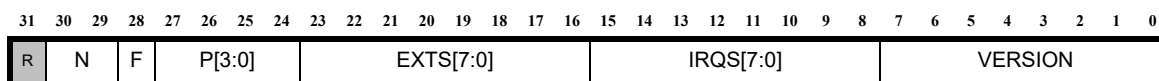


Table 4-45 IRQ_BUILD Field Descriptions

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



The reset value of this register is defined by the configuration of the interrupt controller, which sets the number of interrupts and the number of those interrupts that are externally visible as interrupt lines. If the non-maskable imprecise exception is configured without a cluster component, but the number of configured interrupts is 0, then the IRQ_BUILD register contains the value 0x20000002.

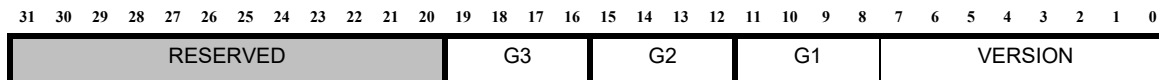
If the non-maskable imprecise exception is configured with a cluster component, but the number of configured interrupts is 0, then the IRQ_BUILD register contains the value 0x40000002.

4.5.12 ARCV2 ISA Profile Register, ISA_PROFILE

Address: 0xF8

Access: R

Figure 4-59 ISA_PROFILE Register



This register indicates the profile that is used as the top-level configuration for the core. Each profile defines a set of available extension options that have overlapping instruction encodings, and are therefore to mutually exclusive to some extent. [Table 4-46](#) describes the three opcode groups that can be configured in a processor.

Table 4-46 ISA_PROFILE field descriptions

Field	Bit	Description
VERSION	[7:0]	ISA Version. All non-enumerated values are reserved. 0x1: Current version
G1	[11:8]	Group 1 ISA extension options. All non-enumerated values are reserved. 0x0: No extension instructions are enabled. 0x1: Code density 1 instructions are enabled (SETEQ, SETNE, SETLT, SETGE, SETLO, SETHS, SETLE, SETGT, ENTER_S, LEAVE_S, BI, BIH LDI, LDI_S) 0x2 to 0x15: Reserved
G2	[15:12]	Group2 ISA extension options. All non-enumerated values are reserved. 0x0: No extension instructions are enabled. 0x1: Code density 2 instructions (LD_S, LDB_S, LDH_S, ST_S, STB_S, and STH_S) are enabled. 0x2: The vector unit instructions and Code density 2 instructions are enabled, and share the opcodes: 0x11, 0x12, 0x13, 0x15, 0x16 and 0x1F. 0x3 to 0x15: Reserved

Table 4-46 ISA_PROFILE field descriptions (Continued)

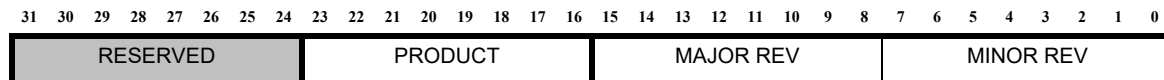
Field	Bit	Description
G3	[19:16]	Group3 ISA extension options. ■ 0x0: No extension instructions are enabled. ■ 0x1: Code density 3 instructions (JLI_S and EI_S) are enabled. ■ 0x2: ARC64 and Code density 3 instructions are enabled, and share the opcode: 0x0B. ■ 0x3 to 0x15: Reserved
RESERVED	[31:20]	Reserved. Read-as-zero; ignored on write.

4.5.13 Micro Architecture Configuration Register, MICRO_ARCH_BUILD

Address: 0xF9

Access: R

Figure 4-60 MICRO_ARCH_BUILD Register



This register indicates the micro-architecture implementation (product family) information to the TCF flow. Read this register to determine the release revision for a product family. This register is used to track all the patch and maintenance releases through its major and minor fields. See the Release Notes in the ARCV2-processor databook to determine the feature updates.

Table 4-47 MICRO_ARCH_BUILD Field Descriptions

Field	Bit	Description
MINOR REV	[7:0]	This field indicates the minor revision number for the product indicated in the [23:16] field. This field is incremented for every patch release.
MAJOR REV	[15:8]	This field indicates the major revision number for the product indicated in the [23:16] field. This field is incremented on every major or maintenance release.
Product Family	[23:16]	This field indicates the product family. All non-enumerated values are reserved. 0x1: ARC EM family 0x2: Reserved 0x3: ARC HS3x family, ARC VPX5 family, and EV family 0x4: ARC HS4x family 0x5-0xFF: Reserved
RESERVED	[31:24]	Reserved. Read-as-zero; ignored on write.

5

Interrupts and Exceptions

5.1 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.

5.2 Privileges and Operating Modes

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

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

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

5.2.1 Kernel Mode

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.

5.2.2 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.

5.2.3 Privilege Violations

This section describes the privileges available to the tasks running in user mode and kernel mode. [Table 5-1](#) gives an overview of the differences in privilege between the two modes.

Table 5-1 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 $\neq$ User PID	If global bit set	If global bit set
Unprivileged instructions	Yes	Yes
Privileged instructions	No	Yes
Access to Basecase Auxiliary Registers	Only LP_START, LP_END, PC and STATUS32[ZNCV]	Yes
Access to extension auxiliary registers	JLI_BASE, LDI_BASE and EI_BASE	Yes
Build Configuration Registers	No	Yes
Timer access	No	Yes
TRAP_S n, SWI, SWI_S	Yes	Yes
Interrupt Enable, level selection	No	Yes
Extension instructions and state	If permitted by XPU	Yes

5.2.3.1 Privileged Instructions

All ARCV2 instructions are unprivileged unless specifically defined as privileged. Privileged instructions are:

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

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

- [BRK](#)

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

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

5.2.3.2 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 kernel mode.

5.2.4 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:

- [TRAP_S](#), [SWI](#), [SWI_S](#)
- Interrupt
- Exception
- [Reset](#) or Machine check exception

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

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

Exception and interrupt handlers may choose to adjust the values in their return address (see [Exception Return Address, ERET](#), [Link Registers](#), [ILINK \(r29\)](#), [BLINK \(r31\)](#)) and status link registers (see [Exception 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).

5.3 Interrupts

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

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

This section explains the following topics:

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

5.3.1 Interrupt Unit Features

ARCV2 interrupt unit has the following interrupt specifications:

- Support for up to 240 interrupts
 - User configurable from 0 to 240
 - Level sensitive or pulse sensitive
- The ARCV2 ISA permits up to 16 interrupt levels; individual ARCV2-based processors may implement fewer levels. For precise information for your ARCV2-based processor, see the processor *Databook*.
 - Programmable from 0 (highest priority) to 15 (lowest priority)
- The priority of each interrupt can be programmed individually by software
- Interrupts can be preempted by higher-priority interrupts
 - Optionally, highest priority level 0 interrupts can be configured as 'Fast Interrupts'
 - Optional second core register bank for use with [Fast Interrupts](#) option to minimize interrupt service latency by minimizing the time needed for context saving
- Automatic save and restore of selected registers on interrupt entry and exit for fast context switch
- User context saved to user or kernel stack, under program control

- Software can set a priority level threshold in STATUS32.E that must be met for an interrupt request to interrupt or wake the processor
- Minimal interrupt / wake-up logic clocked in sleep state
 - Interrupt prioritization logic is purely combinational
- Any Interrupt can be triggered by software

5.3.2 Interrupt Unit Configuration

You can configure the interrupt controller using the options in [Table 5-2](#):

Table 5-2 Interrupt Configuration Options

Configuration Option	Range	Description
has_interrupts	false, true	Indicates if the processor configuration includes the interrupt unit. false—indicates that the interrupt unit is not included. true—indicates that the interrupt unit is included.
number_of_interrupts	0 to 240 interrupts	Defines the total number of interrupts in the interrupt controller.
number_of_levels	1 to 16	Defines the number of interrupt priority levels (0 to 15) in the interrupt controller.
external_interrupts	up to 240	Defines the total number of interrupt pins available for external system components. This parameter must be less than the total number of interrupts. Additional external interrupts are available when ARConnect IDU is configured.
firq_option	false, true	Indicates whether the fast interrupt option is enabled. false: indicates that the fast interrupt option is disabled. true: indicates that the fast interrupt option is enabled.
intvbase_ext	false, true	Indicates if the interrupt vector base can be set using the external pins on reset. false: the interrupt vector base cannot be set using the external pins on reset. true: the interrupt vector base can be set using the external pins <code>intvbase_in</code> on reset.



Note

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

5.3.2.1 Interrupt Configuration Constraints

The ARCV2 ISA includes the following interrupts:

- Fixed core interrupts: optional interrupts with preassigned vector positions
 - irq16: Timer 0 if configured
 - irq17: Timer 1 if configured
 - irq18: Watchdog timer if configured
 - irq19: Inter-core interrupt if configured (`-mcip_has_intrpt==true`)
 - irq20: Interrupt from the performance counter unit if configured; if the performance counter unit is included in the build and the performance counter unit is enabled to generate interrupts (`-pct_interrupt==true`)
 - irq21: The interrupt from a DMA client is connected to irq21 of the core. This interrupt is a level-sensitive interrupt output from the core.
 - Two interrupts are assigned to the STU if configured; irq21 for the error interrupt signal from the STU and irq22 for the done interrupt signal from the STU.
 - When the vector DSP unit is configured, only two interrupt levels are supported.
- Shared common interrupts: external interrupts distributed to each core by the IDU
 - irq24 to IRQ (`23+mcip_idu_cirq_num`): In multi-core configurations, number of common external interrupts configured (`-mcip_idu_cirq_num`) to use with the ARConnect Interrupt Distribution Unit (IDU); these external interrupts are distributed by the IDU to each core and are connected to sequential interrupt lines on each core: irq24, irq25, and so on. A maximum of 128 common interrupts can be configured for use with IDU.
- External interrupts: core specific (private) interrupt
 - Number of external interrupt pins that you can connect to your components. The maximum number of private (core-specific) external pins can be computed as follows: (Number of interrupts – fixed core interrupts – shared common interrupts – safety error interrupt). External interrupts are assigned after all internal sources (including Safety Error Interrupt) are assigned, allocating the lowest available unused interrupt vectors first. For example, if only TIMER1 is present, external interrupts are assigned vectors 16, 18, 19, and so on.
- Software-only triggered interrupts
 - Software-only triggered interrupts, if any, are assigned vectors starting just above the external interrupts.

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

5.3.2.2 Interrupt Vectors

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

[Table 5-3](#) lists the vector offsets. One 32-bit word is reserved for each interrupt line to allow room for vector address. Vectors are fetched in instruction space and thus may be present in ICCM, Instruction Cache, or main memory accessed by instruction fetch logic.

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

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

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

5.3.2.3 Interrupt Vector Base Address Configuration

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

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

5.3.3 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.

5.3.3.1 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 $(\text{NUMBER\_OF\_LEVELS}-1)$) to the `IRQ_PRIORITY` register. By default, `IRQ_PRIORITY` is set to 0.
 - 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.

5.3.3.2 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.

5.3.3.3 Disabling Interrupts

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

5.3.3.4 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.

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

5.3.3.5 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.

5.3.3.6 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 $[0, \text{NUMBER\_OF\_LEVELS}]$. The priority of an interrupt is set in the `IRQ_PRIORITY` register associated with every interrupt and is in the range $[0 \text{ to } \text{NUMBER\_OF\_LEVELS}-1]$.

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.



Note

Delay-slot instruction can not be interrupted.



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.

5.3.3.7 Banked Interrupt Registers

An interrupt controller supporting M interrupts ($1 \leq M \leq 240$), has M banks of interrupt-specific auxiliary registers, selected by [Interrupt Select, `IRQ\_SELECT`](#). A processor must write to `IRQ_SELECT` before reading or writing to one of the banked registers.

The following are the set of registers that are replicated for each interrupt selected by the `IRQ_SELECT` register:

- **Interrupt Priority Register, IRQ_PRIORITY** – This register defines the priority level of the interrupt selected by the IRQ_SELECT register. A priority register with a value $\geq N$ confers priority $N-1$; where N is the number of interrupt priorities. If multiple asserted interrupts are programmed at the same priority level, the interrupt with the lowest number is given the highest priority.
- **Interrupt Enable Register, IRQ_ENABLE** – When this register is set to 1, the interrupt selected by the IRQ_SELECT register is enabled.
- **Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL** – Pulse-triggered interrupts are generally auto clearing. If you choose not to take a pulse-triggered interrupt, you can use this banked register to clear that pulse-triggered interrupt for the IRQ selected by the IRQ_SELECT register. This register is selected by the value in the IRQ_SELECT register. This register is available only for external interrupts.
- **Interrupt Trigger Register, IRQ_TRIGGER** – This register allows software to program whether the interrupt is level-sensitive or pulse-sensitive by writing 0 or 1 respectively. This register is available only for external interrupts.
- **Interrupt Pending Register, IRQ_PENDING** – When this register is set to 1, it indicates that the interrupt, selected by the IRQ_SELECT register, is pending.
- **Interrupt Status Register, IRQ_STATUS** – This register allows software to read the combined values in IRQ_PRIORITY, IRQ_PENDING, IRQ_ENABLE and IRQ_TRIGGER using a single status register for the selected interrupt. This register has $n+3$ bits.

There is a single IRQ_PRIORITY_PENDING[15:0] register, which provides visibility of whether there is any interrupt pending at each priority level.

Reading or writing to a banked interrupt register is a two-stage process. Interrupts have to be disabled during this process to avoid the IRQ_SELECT register being modified by a separate thread. For banked interrupt registers that are unimplemented, writes are ignored, and reads return zero.

[Example 5-1](#) 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 5-1 Setting Interrupt Priority or Enable (STATUS32.IE) of an Interrupt

```
CLRI  R0                ;;save interrupt enable and then disable
SR    R1, [IRQ_SELECT]  ;;R1 provides the interrupt number
SR    R2, [IRQ_PRIORITY] ;;R2 provides the interrupt priority
SETI  R0                ;;restore previous interrupt enable
```



Note

The CLRI and SETI instructions are serializing instructions.

5.3.3.8 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 5-1](#) 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.

5.3.3.9 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.



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.

5.3.3.10 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.

5.3.3.11 Software-Triggered Interrupts

In addition to the SWI instruction, the interrupt system allows software to generate a specific interrupt by writing to the software interrupt trigger register (see [Software Interrupt Trigger, `AUX\_IRQ\_HINT`](#)). All interrupts can be generated through the `AUX_IRQ_HINT` register (see [Software Interrupt Trigger, `AUX\_IRQ\_HINT`](#)). The [Software Interrupt Trigger, `AUX\_IRQ\_HINT`](#) register can be written by a program running on the processor or from the host.

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.

5.3.3.12 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.

5.3.4 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.

5.3.4.1 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.

5.3.4.2 Fast Interrupts

You can configure the interrupt architecture so that fast interrupts, interrupts with the highest priority level P0, save only PC and STATUS32 registers. The purpose of the fast interrupt option is to avoid memory transactions in the entry and exit sequences for interrupts at level P0. An optional second core register bank may be used so that these registers need not be saved or restored from a stack when servicing a fast interrupt. Use the following option to configure fast interrupts:

`FIRQ_OPTION`

The following is the behavior of fast interrupt option:

When `FIRQ_OPTION` is 1, the mechanism to push and pop PC, STATUS32, and general purpose registers to the stack on interrupt entry or exit is disabled for level P0 interrupts. When the fast interrupt option is enabled, the STATUS32 register is stored to STATUS32_P0 and the PC register is stored to ILINK during the lifetime of a P0 interrupt handler.

When the `FIRQ_OPTION` is 0, the STATUS32_P0 register is read as zero and ignored on writes, and P0 priority level interrupts behave the same way as other priority levels; context is saved and restored from a stack in memory. Also, when the `FIRQ_OPTION` is 0, the ILINK register is used for temporary storage by the hardware during interrupt entry and exit, and is not usable as a general purpose register.

When `FIRQ_OPTION` is 1 and `RGF_NUM_BANKS` $\geq$ 2, the interrupt entry sequence sets the register bank number STATUS32.RB to 1, the second register bank, when entering a priority P0 interrupt. Similarly, on P0 interrupt exit, the processor returns STATUS32.RB to its previous value by restoring STATUS32.RB from

STATUS32_P0. This configuration causes an automatic switch of the duplicated register banks, allowing all P0 interrupts to use a private subset of registers.

**Note**

`firq_option==true` does not necessarily require that `RGF_NUM_BANKS > 1`.

5.3.4.2.1 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 kernel stack in memory when taking non-P0 interrupts (when `firq_option==true`) or for all interrupts (when `firq_option==false`).

5.3.4.2.2 Fast Interrupt Entry

The following example explains the sequence of a fast interrupt entry. To understand the pseudo code, refer to the following notation:

- P is the processor's current operating priority defined by the index of least significant bit set (with value of 1) in `AUX_IRQ_ACT.ACTIVE`. P is 16 if no interrupts are active.
- Q is the index of the least significant bit set (with a value of 1) in `AUX_IRQ_ACT.ACTIVE` when bit P is cleared. Q is 16 if P is the only bit set to 1 in `AUX_IRQ_ACT.ACTIVE`.
- PI is the highest priority pending and enabled interrupt, and its priority is W and W=0.

Figure 5-1 Fast Interrupt Entry

```
EnterFastInterrupt(PI) {
    // Assert ((P != 0) && (STATUS32.IE) && (STATUS32.E ? 1)
    // && (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
    ILINK = PC;
    STATUS32_P0 = STATUS32;
    // save user-mode flag
    if (AUX_IRQ_ACT.ACTIVE == 0) {
        AUX_IRQ_ACT.U = STATUS32.U
    }
    AUX_IRQ_ACT.ACTIVE[W] = 1;
    if (RGF_NUM_BANKS > 1) {
        STATUS32.{RB=1}
    }
    // switch to kernel SP if previously in user thread
```

```

if (STATUS32.U) {
    AEX R28, [AUX_USER_SP]
}
STATUS32.{Z = U; U = 0; L = 1; ES = 0; DZ = 0; DE = 0;}
AcknowledgeInterrupt(PI);
Jump(InterruptVector(PI));
}

```

5.3.4.2.3 Fast Interrupt Exit

The following pseudo-code explains the sequence of a fast interrupt exit:

Figure 5-2 Fast Interrupt Exit

```

ReturnFromFastInterrupt (PI, W) {
    // Assert ((P == 0) && (STATUS32.AE == 0))
    //if another fast interrupt is pending and enabled
    if ((W == 0) && (W ≤ STATUS32.E) && (STATUS32.IE == 1)) {
        //acknowledge interrupt
        AcknowledgeInterrupt(PI);
        //jump to another fast interrupt vector
        Jump(InterruptVector(PI));
    }
    //else restore context and return from interrupt
    else
    {
        //clear the currently-active interrupt
        AUX_IRQ_ACT.ACTIVE[P] = 0;
        //if (returning to a user thread)
        if ((AUX_IRQ_ACT.ACTIVE == 0) && AUX_IRQ_ACT.U) {
            //restore user SP
            AEX R28, [AUX_USER_SP];
            AUX_IRQ_ACT.U = 0;
        }
        //restore STATUS32
        STATUS32 = STATUS32_P0;
        //jump back to outer context
        Jump(ILINK);
    }
}

```

5.3.4.3 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.

5.3.4.3.1 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 4-3, 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.

- If $STATUS32.U == 1$, the user or kernel stack pointers are exchanged; Therefore, if $AUX\_IRQ\_CTRL.U == 1$, registers are pushed to the stack of the current user stack when a user thread is interrupted, or to the kernel stack when a kernel thread or interrupt handler is interrupted.
- If $AUX\_IRQ\_CTRL.U == 0$ on interrupt entry, the saved registers are saved to the kernel stack regardless of operating mode and current interrupt level.
- When a user thread is interrupted (or an exception occurs), and $AUX\_IRQ\_CTRL.U == 1$, the user and kernel stack pointers are switched automatically after registers have been saved to the user stack. On return to a user thread, from either an interrupt or an exception, when $AUX\_IRQ\_CTRL.U == 1$, the kernel and user stack pointers are switched automatically before core registers are restored to the user stack. However, when $AUX\_IRQ\_CTRL.U == 0$, any stack pointer switching takes place before state is pushed, and after state is popped. In this case, the kernel stack is used.



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.

5.3.4.3.2 Exceptions During Interrupt Entry and Exit

Memory operations can take place during the automated saving and restoration of CPU context during interrupt entry and exit sequences. The interrupt entry and exit sequences are also referred to as prolog and epilog sequences, respectively. These memory operations can themselves raise memory-related exceptions, depending on the configuration. It is therefore possible to enter an exception handler as a result of attempting to automatically save CPU context prior to interrupt entry. It is also possible to enter an exception handler while restoring CPU context during the execution of an RTIE instruction.

The ARCV2 architecture guarantees that the interrupt entry or exit sequence can be replayed correctly.

The interrupt entry and exit pseudo code, describe the automatic save and restore of core registers. It is important to note that this code is not meant to imply the order in time of saving and restoring registers; only the layout in memory after a sequence successfully completes.

If an exception occurs midway through an prolog or epilog sequence, the user can make no assumption about which transfers to memory are complete as the stack area being written is considered to be unallocated until the sequence has committed, at which point the stack pointer is updated.

Following a premature termination of a prolog or epilog sequence, after the exception is resolved and the exception handler executes the RTIE instruction, the failing prolog or epilog sequence is retried from the beginning.

The AUX_IRQ_CTRL.U bit, together with STATUS32.U, determines whether the user-mode stack pointer or the kernel-mode stack pointer is used during interrupt entry and exit sequences to save and restore CPU context. All memory operations initiated within interrupt entry or exit sequences, are performed with kernel privilege if using the kernel-mode stack. Likewise, the memory operations are performed with user privilege if using the user-mode stack.

You should ensure that stack pointers are 32-bit aligned. The ARCV2 application binary interface (ABI) defines the pointers to be 32-bit aligned and ARCV2 implementations take advantage of this interface to minimize interrupt service latency. During execution of the interrupt prolog or epilog sequence, if the stack pointer is not 32-bit aligned, a misaligned access exception is always raised irrespective of the STATUS32.AD bit.

5.3.4.3.3 Regular Interrupt Entry

The following pseudo-code explains the sequence of a regular interrupt entry. To understand the pseudo code, refer to the following notation:

- P is the processor's current operating priority defined by the index of the least significant bit set (with value of 1) in AUX_IRQ_ACT.ACTIVE. P is 16 if no interrupts are active.
- Q is the index of the least significant bit set (with a value of 1) in AUX_IRQ_ACT.ACTIVE when bit P is cleared. Q is 16 if P is the only bit set to 1 in AUX_IRQ_ACT.ACTIVE.
- PI is the highest priority pending and enabled interrupt, and its priority is W.
-

Example 5-2 Regular Interrupt Entry

```
EnterInterrupt(PI, W) {
    // Assert ((W < P) && (STATUS32.IE) && (W = STATUS32.E)
    // && (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
```

```

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;
Tmp_addr = SP - (4 * ( (1 * (AUX_IRQ_CTRL.B == 1))
                      + (3 * (AUX_IRQ_CTRL.L == 1))
                      + (2 * (AUX_IRQ_CTRL.LP & CD2_OPTION))
                      + (1 * (AUX_IRQ_CTRL.LP & JLI_OPTION))
                      + (2 * (((RGF_NUM_REGS == 16) && (AUX_IRQ_CTRL.NR > 8)) ? 8 : NR))
                      + 2));

Tmp_SP = Tmp_addr;
if (RGF_NUM_REGS == 32) {
    for (i = 0; i < (2 * AUX_IRQ_CTRL.NR); i += 2) {
        Store (Ri, Tmp_addr);
        Tmp_addr = Tmp_addr + 4;
        Store (Ri+1, Tmp_addr);
        Tmp_addr = Tmp_addr + 4;
    }
    if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16)) {
        Store (BLINK, Tmp_addr);
        Tmp_addr = Tmp_addr + 4;
    }
}
// (RGF_NUM_REGS == 16)
else {
    if (AUX_IRQ_CTRL.NR >= 1) {
        Store( R1, Tmp_addr); Tmp_addr += 4;
        Store( R0, Tmp_addr); Tmp_addr += 4;
    }
    if (AUX_IRQ_CTRL.NR >= 2) {
        Store( R2, Tmp_addr); Tmp_addr += 4;
        Store( R3, Tmp_addr); Tmp_addr += 4;
    }
    if (AUX_IRQ_CTRL.NR >= 3) {
        Store( R11, Tmp_addr); Tmp_addr += 4;
        Store( R10, Tmp_addr); Tmp_addr += 4;
    }
    if (AUX_IRQ_CTRL.NR >= 4) {
        Store( R13, Tmp_addr); Tmp_addr += 4;
        Store( R12, Tmp_addr); Tmp_addr += 4;
    }
    if (AUX_IRQ_CTRL.NR >= 5) {
        Store( R15, Tmp_addr); Tmp_addr += 4;
        Store( R14, Tmp_addr); Tmp_addr += 4;
    }
}

// save ZOL state
if (AUX_IRQ_CTRL.L) {

```

```

        store(LP_END, Tmp_addr); Tmp_addr += 4;
        store(LP_START, Tmp_addr); Tmp_addr += 4;
        store(LP_COUNT, Tmp_addr); Tmp_addr += 4;
    }
// save Code Density State
if (AUX_IRQ_CTRL.LP && (CD2_OPTION || JLI_OPTION)) {
    if (CD2_OPTION == 1) {
        store(EI_BASE, Tmp_addr); Tmp_addr += 4;
        store(LDI_BASE, Tmp_addr); Tmp_addr += 4;
    }
    if (JLI_OPTION == 1) {
        store(JLI_BASE, Tmp_addr); Tmp_addr += 4;
    }
}
Push(STATUS32);
Push(PC);

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; }
AcknowledgeInterrupt(PI);
Jump(InterruptVector(PI));
}

```

**Note**

The code in this example does not imply the order in time of memory operations; it only implies the layout in the memory. For more information, see [Exceptions During Interrupt Entry and Exit](#).

5.3.4.4 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.

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.

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.

5.3.4.4.1 Regular Interrupt Exit

The following pseudo-code explains the sequence of a regular interrupt exit:

The following example explains the sequence of a regular interrupt entry. To understand the pseudo code, refer to the following notation:

- P is the processor's current operating priority defined by the index of the least significant bit set (with value of 1) in AUX_IRQ_ACT.ACTIVE. P is 16 if no interrupts are active.
- Q is the index of the least significant bit set (with a value of 1) in AUX_IRQ_ACT.ACTIVE when bit P is cleared. Q is 16 if P is the only bit set to 1 in AUX_IRQ_ACT.ACTIVE.

- PI is the highest priority pending and enabled interrupt, and its priority is W.

Definitions of the functions used are provided at the end of the example.

Example 5-3 Regular Interrupt Exit

```
ReturnFromInterrupt (PI, W){
    //Assert(((W >= P) || (W < STATUS32.E)) && ((P > 0) || (FIRQ_OPTION ==0)) &&
    //(STATUS32.AE == 0))
    // clear the currently-active interrupt
    AUX_IRQ_ACT.ACTIVE[P] = 0;
    // if pending interrupts, avoid pop/push
    if (((W > 0) || (FIRQ_OPTION == 0)) && ((W < Q) && (W ? STATUS32.E))) {
        // set new interrupt level
        AUX_IRQ_ACT.ACTIVE[W] = 1;
        STATUS32.L = 1;
        STATUS32.Z = U;
        STATUS32.{Z = U; U = 0; L = 1; ES = 0; DZ = 0; DE = 0;}
        // acknowledge interrupt
        AcknowledgeInterrupt(PI);
        // jump to interrupt vector
        Jump(InterruptVector(PI));
    }
    // else pop context before interrupt exit
    else {
        // if returning to user thread and user stack stores user context
        if ( (AUX_IRQ_ACT.ACTIVE == 0) && AUX_IRQ_ACT.U && AUX_IRQ_CTRL.U) {
            AEX R28, [AUX_USER_SP];
            STATUS32.U = 1;
        }
        // If the register bank has 32 entry registers files
        if (RGF_NUM_REGS == 32) {
            // pop AUX_IRQ_CTRL.NR register pairs
            for (i = 0; i < (2 * AUX_IRQ_CTRL.NR); i += 2){
                if (i == 28) {
                    /* not restore SP */
                    Pop(tmp);
                    Pop(Ri+1);
                }
                else{
                    Pop(Ri);
                    Pop(Ri+1);
                }
            }
            // if BLINK should be restored separately
            if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16)){
                Pop(BLINK);
            }
        }
    }

    // If the register bank has 16 entry register files
    else {
        if (AUX_IRQ_CTRL.NR >= 1) { Pop( R0); Pop( R1); }
    }
}
```



```

        if (AUX_IRQ_CTRL.NR >= 2) { Pop( R2); Pop( R3); }
        if (AUX_IRQ_CTRL.NR >= 3) { Pop(R10); Pop(R11); }
        if (AUX_IRQ_CTRL.NR >= 4) { Pop(R12); Pop(R13); }
        if (AUX_IRQ_CTRL.NR >= 5) { Pop(R14); Pop(R15); }
        if (AUX_IRQ_CTRL.NR >= 6) { Pop(R26); Pop(R27); }
        if (AUX_IRQ_CTRL.NR >= 7) { Pop(R28); Pop(R29); }
        if (AUX_IRQ_CTRL.NR >= 8) { Pop(R30); Pop(R31); }
        if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 8)) {
            Pop(BLINK);
        }
    }
// restore ZOL state
if (AUX_IRQ_CTRL.L) {
    Pop(LP_END);
    Pop(LP_START);
    Pop(LP_COUNT);
}
// restore Code Density State
if (AUX_IRQ_CTRL.LP && (CD2_OPTION || JLI_OPTION)) {
    if (CD2_OPTION == 1) {
        Pop(EI_BASE);
        Pop(LDI_BASE);
    }
    if (JLI_OPTION == 1) {
        Pop(JLI_BASE);
    }
}
// pop PC
Pop(PC);
// pop STATUS32
Pop(STATUS32);
AUX_IRQ_ACT.U = 0;

// if returning to user thread and kernel stack stores user context
if ( (AUX_IRQ_ACT.ACTIVE == 0) && AUX_IRQ_ACT.U && !AUX_IRQ_CTRL.U) {
    AEX R28, [AUX_USER_SP];
    AUX_IRQ_ACT.U = 0;
}
// jump back to outer context
Jump(PC);

```

**Note**

The code in this example does not imply the order in time of memory operations; it only implies the layout in the memory. For more information, see [Exceptions During Interrupt Entry and Exit](#).

5.3.4.5 Interrupt Timing

Interrupts are held off when a compound instruction has a dependency on the following instruction or is waiting for immediate data from memory. This dependency occurs during a branch, jump, or simply when an instruction uses long immediate data. The time taken to service an interrupt encompasses a jump to the appropriate vector, and then a jump to the routine pointed to by that vector. The timings of interrupts, according to the configuration of the processor, are defined by each ARCV2 processor Databook.

The time taken to service an interrupt also depends on the following:

- Whether an I- Cache miss occurs when accessing the interrupt vector table or when fetching the instructions at the entry point of the service routine
- The number of registers saved to a software stack at the start of the interrupt service routine
- Whether an interrupt of the same or higher level is already being serviced
- Whether the interrupt is itself interrupted by a higher level interrupt or an exception

5.3.4.6 Determine Active Interrupts

The following pseudo-code demonstrates how to determine the active interrupts in a processor:

Figure 5-3 Determine Active Interrupts

```
scan_interrupt_vectors()
{
    CLRI R0 ;; disable interrupts before manipulating IRQ_SELECT
    for all interrupt numbers x from 0 to 255
    do {
        write x to the IRQ_SELECT register
        write x to the AUX_IRQ_HINT register
        read from the IRQ_STATUS register, and if the IRQ_STATUS.IP bit (31) is set to 1,
        vector x is an interrupt
        Otherwise vector x is not an interrupt.
        If x < 16, x is an exception vector, otherwise x is beyond the defined interrupt
        region.
    }
    write 0 to the AUX_IRQ_HINT register to clear all software interrupt hints
    SETI R0 ;; re-enable interrupts
}
```

5.4 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.

5.4.1 Exception Precision

The term **precise exception** refers to a synchronous exception event associated with a specific instruction. When a precise exception is taken, the exception handler knows precisely the point in the program at which the exception took place. If a precise exception can be restarted, all state changes from instructions that occurred before the exception point are completed, and none of the state changes from instructions that occur after the exception point are completed.

All ARCV2-based processors implement precise exceptions, and all instructions can be restarted up and until they are committed. Apart from instructions that retire post-commit, instructions can always be

dismissed before completion and restarted later. On receipt of an exception, an operating system can do any of the following:

- Kill the process
- Send a signal to the process
- Intervene to remove the cause of the exception, and restart the process

A memory error exception may not be recoverable depending on the actual event that caused the memory error. For example:

- An instruction cache load that causes a bus error, and therefore a [Machine Check, Internal Instruction Memory Error](#), is precise because the address of the instruction is known at the time of the memory error.
- A load or store operation that causes a bus error, and therefore a [Bus Error on Data Access to External Data Memory](#) exception, may be imprecise if the processor is configured with a data cache that has a copy-back policy. In such cases, a copy-back operation may fail when writing a block from cache to an address that originated from an earlier load or store instruction. If this copy-back operation occurs, the program would have progressed beyond the originating load or store instruction, and therefore, the exception is imprecise and irrecoverable.

Imprecise exceptions are asynchronous events that may or may not be associated with a specific instruction. When an imprecise exception is raised and the processor is in exception mode, the first such delayed exception is taken after the processor has exited the Exception mode. Any second or subsequent imprecise exceptions are discarded.



Note

- When an imprecise exception is raised and the processor is in the exception mode, the first such delayed exception is taken after the processor has exited the Exception mode. However, in this situation imprecise exceptions raised by the vector unit (EV_VecUnit exception or EV_MachineCheck exception with cause code 0x80) are not delayed, but are instead converted to an EV_MachineCheck double fault exception.
- The exception handler of imprecise exceptions must have DSYNC as the first instruction of the exception handler. This design ensures that all outstanding writes prior to the exception have completed before any new write (in the exception handler) commits. This includes the handler for the following imprecise exceptions: memory errors (including bus errors), double-bit ECC errors, and imprecise floating-point exceptions.

5.4.1.1 Non-Maskable Imprecise Exception

You can trigger a high priority non-maskable imprecise exception by asserting the external signal: `nmi`. Use the `-nmi_option` to configure this support. When the `nmi` signal is asserted, a non-maskable imprecise exception is raised with ECR 0x031400 cause code. You can use this exception in situations where non-correctable errors need immediate attention and situations where you want to restore the core from lockup status and read the debug information.

In a multi-core cluster, when the `nmi` signal is asserted high, a non-maskable imprecise exception is triggered for each core that is not in a halted or powered down state. If in a multi-core cluster, a core is stalled and other cores are not stalled, use the following process to reset the stalled core using `nmi`:

1. Halt all the cores that are not stalled.

2. Assert the `nmi` signal to reset the stalled cores.
3. Wait for the `nmi` handshake to complete.
4. Restore the halted cores to the run state.

Related Topics:

- See [Exception Types and Priorities](#) for information on the non-maskable imprecise exception priority
- See [Machine Check, Non-Maskable Imprecise Exception, Double fault](#)

5.4.2 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

5.4.2.1 Vector Name

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

5.4.2.2 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.

5.4.2.3 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 5-3](#).

Table 5-3 Interrupt/Exception Vectors

Name	Vector Offset	Vector Number	Exception Types
Reset	0x00	0x00	Exception
Memory Error	0x04	0x01	Exception
Instruction Error	0x08	0x02	Exception
EV_MachineCheck	0x0C	0x03	Exception
EV_TLBMissI	0x10	0x04	Exception
EV_TLBMissD	0x14	0x05	Exception

Table 5-3 Interrupt/Exception Vectors (Continued)

Name	Vector Offset	Vector Number	Exception Types
EV_ProtV	0x18	0x06	Exception
EV_PrivilegeV	0x1C	0x07	Exception
EV_SWI	0x20	0x08	Exception
EV_Trap	0x24	0x09	Exception
EV_Extension	0x28	0x0A	Exception
EV_DivZero	0x2C	0x0B	Exception
EV_DCError	0x30	0x0C	Exception
EV_Misaligned	0x34	0x0D	Exception
EV_VecUnit	0x38	0x0E	Vector unit exceptions
Unused	0x3C	0x0F	
IRQ16 - IRQ 255	0x40 - 0x3FC	0x10 - FF	Interrupts

The entries in an exception vector table contain addresses of interrupt and exception handlers, not jump instructions to those handlers. Although the processor always correctly jumps to a handler if an exception or an interrupt happens, execution of a jump instruction to the vector table has undefined behavior.

[Table 5-4](#) provides more information about the exception priorities and exception cause parameters.

If you need to explicitly pass control to an interrupt handler, use the `AUX_IRQ_HINT` register.

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

Example 5-4 Pass Control to an Handler or Perform a Soft Reset

1. Read the contents of a the vector table entry into a core register.
2. Execute an indirect jump on the core register.

The following code shows how a soft reset can be implemented:

```

LR %R0, [%INT_VECTOR_BASE]
LD %R1, [%R0]
J [%R1]

```

Example 5-5 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

```

5.4.2.4 Cause Codes

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

5.4.2.5 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.

5.4.2.6 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 4-35](#) on page 168.

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*.

5.4.3 Exception Types and Priorities

Multiple exceptions can be associated with a single instruction, but only one exception can be handled at a time. Remaining exceptions are considered when the instruction is restarted after the first exception handler has completed. This process continues until no further exceptions remain, or a non-recoverable exception is encountered.

Interrupts and exceptions are evaluated with the following priority:

1. Reset
2. Machine Check, Non-maskable imprecise exception double Fault
3. Machine Check: Non-maskable imprecise exception

4. Machine Check, Double Fault
5. Machine Check, Fatal TLB errors
 - a. Overlapping TLB entries
 - b. Illegal Overlapping MPU entries
6. Interrupt
7. TLB Fault on Instruction Fetch
 - a. Machine Check, Memory Error on ITLB Access
 - b. ITLB miss
8. Protection Violation on Instruction Fetch (MMU or MPU)
9. Memory Error on instruction fetch:
 - a. Bus error accessing external instruction memory
 - b. Instruction fetch from unpopulated ICCM region
 - c. Instruction fetch spanning multiple MPU regions
 - d. Instruction fetch spanning multiple instruction memory targets
10. Privilege Violation, Actionpoint hit on Instruction fetch

**Note**

While protection violation on instruction fetch has higher priority than Actionpoint hit on instruction fetch violation, Actionpoint breakpoint halt has higher priority than instruction fetch violation.

11. Machine Check, Internal Instruction Memory Error
12. Instruction Error
 - a. Illegal instruction exception (including Invalid TLB command)
 - b. Illegal instruction sequence exception
13. Privilege Violation
 - a. Access to kernel resource in user mode
 - i. Kernel -only instruction or register violation
 - ii. Access to a secured resource in normal mode
 - b. APEX extension group
 - i. APEX extension group – User mode access disabled by XPU
 - ii. APEX extension group – kernel only extension violation
 - c. Disabled extension group
14. Memory error on XY Access
 - a. Illegal AGU memory target address
 - b. Unaligned AGU address spanning boundary

- c. 64-bit AGU destination address is not 32 bit aligned
- 15. Extension Instruction Exception - requested by extension instruction
- 16. Misaligned data memory access
- 17. TLB fault on data access
 - a. Machine check, memory error on DTLB access
 - b. DTLB miss
- 18. Protection Violation
 - a. Protection Violation on data access (MMU, MPU, Code Protection, Stack Checking, or secure)
 - b. Non-sequential execute target address MPU violations (overlap region, illegal secure to normal transition)
- 19. Memory error on data access:
 - a. Data access to unpopulated DCCM region or absent memory
 - b. Data access spanning multiple data memory targets
- 20. Machine check, internal data memory error
- 21. Divide by zero exception when STATUS32[DZ] = 1
- 22. Trap or software interrupt (TRAP_S, SWI or SWI_S instructions)
- 23. Bus error on data access to external data memory (or ICCM)
- 24. Actionpoint hit on Auxiliary register or Memory-access – an imprecise exception
- 25. Machine check - uncorrectable ECC or parity error in vector memory
- 26. Misaligned vector memory access
- 27. Vector stack pointer checking protection violation
- 28. Vector memory address out of range

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 5-4](#) describes the exception vectors, exception cause codes, and exception parameters.

5.4.3.1 MMU Exception Priorities

When an instruction or data access spans two MMU pages, and there are exceptions raised by both pages, the core first takes the exception from the page with the lowest address.

5.4.3.2 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

5.4.3.3 Exception Vectors and Cause Codes

Table 5-4 indicates the ECR values when an exception is raised. If simultaneous violations occur, the Parameter field in ECR has one bit set for each of those exceptions. For example, if there are simultaneous Code Protection, Stack Checking memory write violations, the Parameter field reads 3 (bit 0 is set for code protection and bit 1 is set for stack checking).

Table 5-5 on page 238 provides a summary of the values placed in the EFA and ERET registers for different exception types.

Table 5-4 Exception Vectors and Cause Codes

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Reset	Reset	0x00	0x00	0x00	0x00	0x000000
Bus Error from Instruction Memory	Memory Error	0x04	0x01	0x00	0x0u	0x01000u
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
Bus Error from Data Memory	Memory Error	0x04	0x01	0x10	0x0u	0x01100u'
Data access from unpopulated ICCM or DCCM region	Memory Error	0x04	0x01	0x11	0x00	0x011100

Table 5-4 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Data access spanning multiple data memory targets	Memory Error	0x04	0x01	0x12	0x00	0x011200
NVM write or erase programming failure	Memory Error	0x04	0x01	0x13	0x00	0x011300
NVM address out of range (instruction fetch or data access) only when NVM is configured	Memory Error	0x04	0x01	0x14	0x00	0x011400
XY access to illegal target memory	Memory Error	0x04	0x01	0x21	0x00	0x012100
XY access spanning multiple targets	Memory Error	0x04	0x01	0x22	0x00	0x012200
XY 64-bit destination is not 32-bit aligned	Memory Error	0x04	0x01	0x23	0x00	0x012300
Illegal Instruction	Instruction Error	0x08	0x02	0x00	0x00	0x020000
Illegal Instruction Sequence	Instruction Error	0x08	0x02	0x01	0x00	0x020100
Double Fault	EV_MachineCheck	0x0C	0x03	0x00	0x00	0x030000
Overlapping TLB Entries	EV_MachineCheck	0x0C	0x03	0x01	0x00	0x030100
Fatal TLB Error	EV_MachineCheck	0x0C	0x03	0x02	0x00	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
Illegal Overlapping MPU entries (jump and branch target)	EV_MachineCheck	0x0C	0x03	0x06	0x01	0x030601

Table 5-4 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Non-Maskable imprecise exception	EV_MachineCheck	0x0C	0x03	0x14	0x00	0x031400
Non-maskable imprecise exception double fault	EV_MachineCheck	0x0C	0x03	0x14	0x01	0x031401
Instruction Fetch TLB Miss	EV_ITLBMiss	0x10	0x04	0x00	0x00	0x040000
Data TLB Miss on read	EV_DTLBMiss	0x14	0x05	0x01	0x00	0x050100
Data TLB miss on write	EV_DTLBMiss	0x14	0x05	0x02	0x00	0x050200
Data TLB miss on RMW	EV_DTLBMiss	0x14	0x05	0x03	0x00	0x050300
Instruction Fetch Protection Violation in MPU	EV_ProtV	0x18	0x06	0x00	0x04	0x060004
Instruction Fetch Protection Violation in MMU	EV_ProtV	0x18	0x06	0x00	0x08	0x060008
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)	EV_ProtV	0x18	0x06	0x01	0x08	0x060108
Memory Write (ST, PUSH, ENTER, interrupt entry, SCND) protection violation in code protection scheme (parameter code 0x01)	EV_ProtV	0x18	0x06	0x02	0x01	0x060201

Table 5-4 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in stack checking scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x02	0x02	0x060202
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in MPU (parameter code 0x04)	EV_ProtV	0x18	0x06	0x02	0x04	0x060204
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in MMU (parameter code 0x08)	EV_ProtV	0x18	0x06	0x02	0x08	0x060208
Memory Write Protection Violation from NVM	EV_ProtV	0x18	0x06	0x02	0x10	0x060210
Memory Read-Modify-Write (EX) protection violation in code protection (parameter code 0x01)	EV_ProtV	0x18	0x06	0x03	0x01	0x060301
Memory Read-Modify-Write (EX) protection violation in stack checking protection scheme (parameter code 0x02)	EV_ProtV	0x18	0x06	0x03	0x02	0x060302
Memory Read-Modify-Write (EX) protection violation in MPU scheme (parameter code 0x04)	EV_ProtV	0x18	0x06	0x03	0x04	0x060304
Memory Read-Modify-Write (EX) protection violation in MMU scheme (parameter code 0x08)	EV_ProtV	0x18	0x06	0x03	0x08	0x060308
Memory Read-Modify-Write Protection Violation from NVM	EV_ProtV	0x18	0x06	0x03	0x10	0x060310
Action Point Hit, Instruction Fetch	EV_PrivilegeV	0x1C	0x07	0x02	0xnn	0x0702nn

Table 5-4 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Parameter	
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
Software Interrupt	EV_SWI	0x20	0x08	0x00	0xnn	0x0800nn
Trap	EV_Trap	0x24	0x09	0x00	0xnn	0x0900nn
Extension Instruction Exception	EV_Extension	0x28	0x0A	mm	0xnn	0x0Ammnn
Invalid Operation, Floating-point extension exceptions	EV_Extension	0x28	0x0A	0x00	0x01	0x0A0001
Divide by Zero, Floating-point extension exceptions	EV_Extension	0x28	0x0A	0x00	0x02	0x0A0002
Divide by zero, Integer exception	EV_DivZero	0x2C	0x0B	0x00	0x00	0x0B0000
Data cache consistency error	EV_DCError	0x30	0x0C	0x00	0x00	0x0C0000
Misaligned data access	EV_Misaligned	0x34	0x0D	0x00	0x00	0x0D0000
Misaligned data access: a LD or ST using an unaligned AGU based access straddles a modulo wrap boundary	EV_Misaligned	0x34	0x0D	0x10	0x00	0x0D1000
Misaligned vector memory access	EV_VecUnit	0x38	0x0E	0x02	0x00	0x0E0200
Vector memory address out of range	EV_VecUnit	0x38	0x0E	0x03	0x00	0x0E0300
Vector stack pointer checking protection violation	EV_VecUnit	0x38	0x0E	0x05	0x00	0x0E0500
Reserved	-	0x3C	0x0F	-		-

The value of 'u' in the parameter field of Bus Error exceptions is determined by the value of STATUS32[U] bit at the time that the instruction triggering the exception was committed. Hence, a User-mode instruction fetch that results in a bus error from external memory, raises an exception with ECR 0x010001. The same exception in Kernel mode has an ECR value of 0x010000.

A load/store access targeting ICCM may be treated in some implementations as an external memory access (external to the Data Memory interface); in this case, if a memory error such as uncorrectable ECC error is encountered in either a load or partial store (RMW) to ICCM, a Bus Error from Data Memory exception may be raised. See “[Bus Error on Data Access to External Data Memory](#)” on page 247.

An example of an extension is floating-point extension. For this extension, the following are the values of sub cause code and parameter:

mm = 0 — indicates floating-point extension exceptions

nn—indicates specific exception(s) raised

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

The rules for prioritizing exceptions are applied normally, when multiple MMU exceptions are raised on two adjacent pages accessed by non-aligned memory operations. That means that the highest priority exceptions are always taken in preference to lower priority exceptions. If two exceptions of the same type occur, for different pages, from the same memory reference, then the exception triggered by the effective byte address (EBA) of the memory operation should be taken first. If a reference spans two pages, then the second address will be EBA+4.

The parameter codes (rr) are implementation dependent. See the processor databook for more information about the error parameter codes for this exception.

Table 5-5 EFA and ERET Entries for Exception Types

Exception	EFA Entry	ERET Entry
Memory access	Address of data access	PC address
Other instruction exceptions	PC address	PC address
Watchpoints	Undefined	Imprecise exception, implementation-dependent
Actionpoint on instruction fetch	PC address	PC address
Bus error caused by uncached load	Data-access address (physical)	Imprecise exception, implementation-dependent
Bus error caused by uncached store	Data-access address (physical)	Imprecise exception, implementation-dependent
Instruction-fetch-related memory exception/ECC parity	PC address	PC address
Bus error reported during data-cache copy-back write	Physical address of the cache line being copied back	Imprecise exception, implementation-dependent
Memory data access ECC or parity error on CCM/data-cache	Data-access address (physical)	Imprecise exception, implementation-dependent
ECC or parity error on data-cache data RAM during copy-back (triggered by LD/ST/EX cache miss or auxiliary operation)	Physical address of the cache line	Imprecise exception, implementation-dependent

Table 5-5 EFA and ERET Entries for Exception Types (Continued)

Exception	EFA Entry	ERET Entry
Memory access ECC or parity error on data-cache tag RAM, DCCM	Data-access address	PC address

5.4.3.4 Reset

A reset is an external reset signal that causes the ARCV2-based processor to perform a *hard* reset. After reset, various internal states of the ARCV2-based processor are preset to their initial values, that is:

1. Interrupts are disabled.
2. The status register flags are cleared.
3. The loop count, loop start, and loop end registers are cleared.
4. The scoreboard unit is cleared.
5. The pending-load flag is cleared.

Program execution begins at the address referenced by the four byte reset vector located at the interrupt vector base address, which is the first entry (offset 0x00) in the vector table. After reset, the vector table base address is set by configuration parameter `INTVBASE_PRESET` (x1024) and reflected as a read-only value in build register [Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD](#). The core registers are not initialized except loop count (which is cleared). The Reset value of vector base register determines the Reset vector address. This reset value is configured at build time through the `INTVBASE_PRESET` parameter. The processor begins execution after reset by fetching the instruction at the address referenced by the reset vector.

A soft reset (an indirect jump to the Reset vector) does *not* preset any of the internal states of the ARCV2-based processor. For more information about how to do a soft reset, see [“Pass Control to an Handler or Perform a Soft Reset”](#) on page 229.

When an MMU is supported and configured in a processor, the `VECBASE_AC_BUILD.ADDR` must be set to an address in the non-translated memory space. This address is within the upper 2 GBytes of the memory space. This design prevents double-fault exceptions from occurring when accessing the vector table to service a first exception. Fetching the vectors is then not subject to TLB misses or other TLB exceptions.

5.4.3.5 Machine Check, Non-Maskable Imprecise Exceptions

A non-maskable imprecise exception is triggered when the `nmi` external signal is asserted high.

In a multi-core cluster, when the `nmi` signal is asserted high, a non-maskable imprecise exception is triggered for each core that is not in a halted or powered down state. In a multi-core cluster, there is only one external `nmi` signal at the cluster.

5.4.3.5.1 Machine Check, Non-Maskable Imprecise Exception, Double fault

A non-maskable imprecise double-fault exception is taken in the following situations:

- The core is handling an exception and a non-maskable imprecise exception is triggered.
- The core is handling a non-maskable imprecise exception and another exception is raised.

In a multi-core cluster, this exception is triggered for each core that satisfies the preceding conditions.

5.4.3.6 Machine Check, Double Fault

Exception detected with exception handler outstanding, as indicated by STATUS32[AE] bit set (see [Status Register, STATUS32](#)).

5.4.3.7 Machine Check, Fatal TLB Error

- Any fatal error in the TLB or its memories (such as a uncorrectable parity or ECC error).

5.4.3.8 Machine Check, Overlapping TLB Entries:

- Multiple matches for an address lookup in the TLB.

5.4.3.9 Protection Violation, Instruction Fetch

An instruction fetch is fetched without the execute permission set (MMU, MPU, code protection, or stack checking).

5.4.3.10 Memory Error on Instruction Fetch

5.4.3.10.1 Bus Error Accessing External Instruction Memory

A bus error exception is raised when an error occurs during a memory access that occurs externally to the unit from which it originated. For example, if an error occurs during an instruction fetch that does not access ICCM or Instruction cache, a bus error is reported. It is the responsibility of the target memory device to detect and report errors, via the appropriate bus interface. These bus errors may be triggered by 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.

5.4.3.10.2 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.

When dynamic loading is configured in the processor, this exception is not triggered; access to unpopulated memory regions are treated as uncached accesses.

5.4.3.10.3 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.

5.4.3.11 Memory Error Exception, Access Spanning Multiple MPU Regions

A memory error exception is raised when access span multiple MPU regions.

5.4.3.12 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

5.4.3.13 Machine Check, Internal Instruction Memory Error

An uncorrectable memory error is detected in one of the internal memories accessed during an instruction fetch. Typically, these errors include ECC double errors, or parity errors, that are detected on instruction fetches from ICCM or Instruction Cache. Memory errors detected during incorrectly speculated accesses are ignored.

For an internal instruction memory error exception, the EFA contains the address of the instruction that contains the unrecoverable error. For an internal data memory error exception the EFA register contains the address of the data item address for which the unrecoverable error occurred.

**Note**

The parameter codes for these errors are implementation-dependent. See the processor databook for information on the ECR codes.

5.4.3.14 Instruction Error

5.4.3.14.1 Illegal Instruction

If an invalid instruction is fetched that the ARCV2-based processor cannot execute, an Illegal Instruction exception is caused. Any use of an unimplemented instruction, condition code, core register, or auxiliary register raises an Illegal Instruction exception. This exception is also raised if a jump or branch instruction with a delay slot specifies long-immediate data for any operand.

A predicated instruction always raises an Illegal Instruction exception if the instruction is encoded illegally, regardless of whether its predicate is true or false. In contrast, a correctly encoded instruction that accesses illegal operands, does not raise an Illegal Instruction exception, if it is predicated and its predicate is false. The false predicate means that the illegal operand is not actually accessed.

Note that a conditional branch instruction with false condition (a not-taken branch) also raises an Illegal Instruction exception if it is incorrectly encoded. Further, a BRcc or BBITn instruction that accesses illegal registers always raises an Illegal Instruction exception, even if its condition turns out to be false; its source registers need to be read to determine the condition, and therefore the operands of BRcc and BBITn are always accessed.

If an instruction specifies a L IMM source operand, and the instruction executes in a delay-slot context (that is, when STATUS32.DE is set to 1), the L IMM value is considered to be unpredictable. Such instructions do not raise an Illegal Instruction exception on the basis of the illegality of the L IMM value itself. However, they raise an Illegal Instruction Sequence exception due to the use of a L IMM source operand by an instruction that is in a delay slot.

For example, consider an AEX.PNZ instruction that accesses an undefined auxiliary register, specified using a core-register operand, when the Z flag is set. In this case the .PNZ predicate is false, and therefore the AEX.PNZ instruction does not raise an Illegal Instruction exception.

Consider also the example of an unconditional AEX instruction, which appears in a delay slot and accesses an undefined auxiliary-register address specified as a L IMM operand. Normally, an access to an unimplemented auxiliary register triggers an Illegal Instruction exception. However, a L IMM auxiliary register address in a delay-slot context is unpredictable and therefore does not trigger an Illegal Instruction exception. This AEX instruction instead raises an Illegal Instruction Sequence exception, due to the presence of its L IMM operand in a delay slot context, as described in [Illegal Instruction Sequence](#) section.

An illegal instruction error can also be caused by an invalid MMU command (SR with invalid command code to TLB Command register). An SR instruction that uses an unimplemented source register, or otherwise causes an Instruction Error exception, does not reach the stage of having its credentials checked and therefore does not raise a TLB error even though the command value is undefined.

5.4.3.14.2 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 `vvaddacc` or `vvsubacc` 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`
- Using a vector register operand to access an un-configured vector register. For example, when you have configured 32 vector registers, a vector register operand that accesses r33 raises an exception.
- An illegal instruction exception is raised when a scalar register used in a vector instruction violates the following rules:

- ❑ Scalar register r60 (LP_COUNT) is used as a scalar source or destination register of a vector instruction.
- ❑ Scalar register r61 is used as source or destination register of a vector instruction, Exception to this rule is when r61 is used as a VK VIMM indicator for operand C of a vector instruction.
- ❑ Scalar register r62 is used as a scalar source or destination register of a vector instruction. Exception to this rule is when r62 is used as a VL VIMM indicator for operand C of a vector instruction.
- ❑ Scalar register r63 (PCL) is read-only.
- ❑ Scalar accumulator registers r58/r59 are used as scalar source operands of a vector instruction.

5.4.3.14.3 Illegal Instruction Sequence

This exception is raised when an Illegal Instruction Sequence is attempted. This exception occurs in the following cases:

- When any branch or jump instruction straddle the loop end position such that:
 - ❑ A branch or jump 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:
 - ❑ A branch or jump instruction (with or without a delay slot and either taken or not taken) (Bcc, BLcc, Jcc, JLcc)
 - ❑ ENTER_S, LEAVE_S, RTIE
 - ❑ EI_S
 - ❑ LPcc
 - ❑ A vector load instruction in the delay slot of a taken branch
 - ❑ Any instruction with long-immediate data as a source operand
- 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:
 - ❑ A branch or jump instruction (with or without a delay slot and either taken or not taken) (Bcc, BLcc, Jcc, JLcc)
 - ❑ ENTER_S, LEAVE_S, RTIE
 - ❑ EI_S
 - ❑ LPcc



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:
 - 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.

- E-slot (the execute slot of an EI_S instruction).

5.4.3.14.4 Vector DSP Illegal Instruction Sequence

- Multiple instructions in a super instruction have the same destination register. A scalar instruction is not allowed to have a destination register that is also an implied scalar address register that is updated by a vector load/store instruction. This rule also applies to the partial overlap of a scalar register with a register pair.
- 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. An additional fourth scalar operand for vvld/vvst address modes is allowed.
- 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. A branch delay slot is not allowed to contain a vector load instruction.
- 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.
- There can be at most one vvset_el_size instruction per super instruction.
- A super instruction bundle can have up to three scalar destination operands for scalar or vector instructions in that super instruction. A scalar instruction in a bundle (that is, SC format bit is set to 1) always counts as having exactly one scalar destination, except for the scalar store, scalar load, PREALLOC, PREFETCH, and PREFETCHW instructions. For these instructions, the actual number of scalar destination registers is counted. For example, ld.ab r0, r1, r2 has two destination

registers; `st r0, r1` has zero destination registers; `prefetch.ab r1, r2` has one destination register; `ld.ab r0, r62, r2` has one destination register (this is a special case that is not normally used); `ld.ab r1, r1, r2` has two destination registers that are the same (this is a special case that is not normally used). A register pair counts as one register. If a vector instruction has the `.f` suffix, there cannot be a scalar instruction bundled with it in the same super instruction, that is, the scalar issue slot must be empty. If a vector instruction has the `.f` suffix, there can not be another vector instruction with a scalar destination operand or `.f` suffix in the same super instruction bundle.

- A vector instruction is not allowed to use the ILINK register (r29) as scalar source operand in a vector-scalar instruction. Violating this rule raises an illegal instruction exception.
- A vector instruction must be scheduled in vector issue slot 0 if it is a vector load or vector store instruction with one or more scalar operands.
- A sequence-type instruction such as VVFFT4 can be bundled with other instructions. The following is a related bundle rule. The dst of an instruction cannot be a src operand of an instruction of class SEQ in the same bundle if that source operand is read after step 0 of the sequence. Violating this rule raise an illegal instruction sequence exception.

5.4.3.15 Privilege Violation

5.4.3.15.1 Privilege Violation, Kernel Only Access

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

5.4.3.15.2 Privilege Violation, Disabled Extension

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

5.4.3.16 Extension Instruction Exception

This exception is triggered when an extension instruction requires that an exception be taken (for example, floating-point extensions need to generate many different types of exceptions).

The following are supplied by the extension instruction:

- mm = subcode
- nn = parameter

Floating-point Extension

mm = 0 — indicates floating-point extension exceptions

nn — indicates specific exceptions raised

nn	=	1 — Invalid Operation floating-point exception; the double-precision floating-point accelerator instructions (DADDx, DSUBx, DMULx, DEXCLx) do not raise this exception.
	=	2 — Divide by Zero floating-point exception; this exception is raised only by the hardware divider. A software divide implementation does not raise this exception.

5.4.3.17 Misaligned Data Access

A misaligned data access raises an EV_Misaligned exception. An access is considered misaligned if the address is not an integer multiple of the operand size and non-aligned references are not supported by the configured core or the STATUS32.AD bit is set to 0. For more information about the rules governing non-aligned memory references, see [Data Layout in Memory](#).

If a LD or ST using an unaligned AGU-based access straddles a modulo wrap boundary in which the buffer size is not a multiple of 16 bytes, a Misaligned memory access exception is raised: vector number 0x0D, vector offset 0x34, cause code 0x10, parameter 0x00. In this case, the exception fault address (EFA) register contains the address that triggered the exception.

5.4.3.18 TLB Fault on Data Access

5.4.3.18.1 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

5.4.3.18.2 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.

5.4.3.19 Instruction Fetch TLB Miss

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

5.4.3.20 Machine Check, Instruction Fetch Memory Error

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

5.4.3.21 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).

5.4.3.22 Memory Error on Data Access

5.4.3.22.1 Memory Error Exception, Access to Unpopulated Memory

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

When dynamic loading is configured in the processor, this exception is not triggered; access to unpopulated memory regions are treated as uncached accesses.

5.4.3.22.2 Memory Error Exception, Access Spanning Multiple Memory Targets

A memory error exception is raised when a memory reference spans two regions that are designated as different memory targets. In this context, a memory target refers to the following types of memory region:

- ICCM
- DCCM

- Data Cache
- External memory (non-volatile memory)
- External memory (volatile memory)
- Peripheral memory
- Translated region
- Untranslated region

This exception cannot happen for aligned data accesses, but might happen for non-aligned data accesses: load and store instructions



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.

5.4.3.23 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.



For load and exchange instructions, when an uncorrectable error occurs, an exception is raised, and the destination register is updated with the uncorrectable data.

5.4.3.24 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.

5.4.3.25 Trap or Software Interrupt

5.4.3.25.1 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.

5.4.3.25.2 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.

5.4.3.26 Bus Error on Data Access to External Data Memory

A bus error exception is raised when an error occurs during a memory access that occurs externally to the unit from which it originated. For example, if an error occurs during a data memory access to an uncached,

non-DCCM location, a bus error is reported. Similarly, if an error occurs during an instruction fetch that does not access ICCM or Instruction cache, again a bus error is reported. It is the responsibility of the target memory device to detect and report errors, via the appropriate bus interface. These bus errors may be triggered by accesses to unpopulated memory regions, or by parity or ECC errors within the external memory devices.

The raising of a bus error exception on an external data memory access may occur after the referencing instruction has committed. For example, if a cache block is written back to memory during cache replacement, and the write transaction experiences a bus error, the fault is caused by a memory address that is different from the one that is currently being accessed by the program.

Because precise exception handling is not possible for external data memory 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.

5.4.3.27 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*.

5.4.3.28 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.

5.4.3.29 Machine Check - Uncorrectable ECC or Parity Error in Vector Memory

5.4.3.30 Double-bit ECC error or single-bit parity error in the vector memory. **Vector Memory Address Out of Range**

The processor checks the addresses of vector load/stores and raise an exception when they don't address the vector memory properly, for example exceed the vector memory size.

The address of any byte of a vector element must be proper. A proper address is either an absolute 32-bit address in the range [AUX_VECMEM_REGION, AUX_VECMEM_REGION + VEC_MEM_SIZE> or a 16-bit or 32-bit address in the range [0, VEC_MEM_SIZE> that is presumed to be an address offset relative to the start of the vector memory.

This is an imprecise exception. The predicate bit must be true for the vector element that is out of range, otherwise the exception is not raised. This exception can be enabled or disabled using the VEC_CTRL[MD] bit.

5.4.3.31 Misaligned Vector Memory Access

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

5.4.3.32 Vector Stack Pointer Checking Protection Violation

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

5.4.4 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.

When an MMU is configured, an instruction or a data item referenced by the instruction operation may span two adjacent pages. In general, 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 is EBA+4.

5.4.5 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.

5.4.6 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.

- ❑ 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 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](#))

3. 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. 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.
4. The exception cause register (see ECR) is loaded with a code to indicate the cause of the exception – see [Table 5-4](#).
5. When a memory access triggers an exception, including a double-fault exception, the exception fault address register (EFA) is loaded with the data item address that triggered the exception. For other faults, the EFA register is loaded with the PC value used to fetch the faulting instruction. For exceptions triggered by breakpoints, the EFA register is loaded with the PC value. Exceptions triggered by watchpoints are imprecise and the EFA register is thus undefined on their occurrence.
6. The CPU is switched into kernel mode STATUS32[U] = 0
7. Interrupts are disabled STATUS32[IE] = 0
8. The “active exception” flag is set. STATUS32[AE] = 1
9. ES (EI table instruction pending) bit is cleared, STATUS32[ES] = 0
10. SC (Stack Checking exception enabled) bit is cleared, STATUS32[SC] = 0
11. DZ (Divide by Zero exception enabled) bit is cleared, STATUS32[DZ] = 0
12. The L bit in STATUS32 is set, disabling ZOL, STATUS32[L] = 1
13. The DE bit in the status register is cleared. STATUS32[DE] = 0
14. 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.

The MMU provides a 32-bit register (SCRATCH_DATA0) that can be used by an operating system to store data.

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.

5.4.6.1 Exceptions During Exception Entry

When an exception occurs during the fetching of an exception vector other than EV_MachineCheck (double fault), then the exception is converted into EV_MachineCheck (double fault) in the usual way.

When an exception occurs during the fetching of an exception vector for an EV_MachineCheck (double fault), a triple fault occurs. In such cases, the DEBUG.TF bit is set to 1, and the processor halts. The

DEBUG.TF bit is available as an external output pin, if implemented, from each core, allowing external hardware to become aware of the fault and take any necessary action.

5.4.7 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.

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 5-6](#).

In [Table 5-6](#), 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 5-6 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 5-6 Exception and Interrupt Exit Modes (Continued)

U	AE	AUX_IRQ_ACT	FIRQ_OPTION	Current Mode	RTIE Response	Link Registers Used
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	N/A

- AE and AUX_IRQ_ACT are all set to 0 for state changes from kernel mode (U bit is 0 in kernel mode), for example when scheduling a user mode task.
- If the AE bit is set, or AE and AUX_IRQ_ACT are all zero, the exception-exit sequence is followed. If AE is zero, and AUX_IRQ_ACT is set to true, the interrupt-exit sequence is followed. See description of the [RTIE](#) instruction for further details.
- The program counter is loaded with the exception return address from the ERET register, the contents of ERSTATUS are copied into STATUS32 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).

5.4.8 Exceptions and Delay Slots

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

Example 5-6 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.

When an exception is detected on a delay-slot instruction, the return address stored on exception entry is normally the address of the instruction in the delay slot. This mechanism allows an exception handler to return to the delay-slot instruction of a taken branch, and for subsequent instructions to be executed starting at the branch target address.

In addition, this mechanism allows branch instructions that can change processor state to have delay slots, for example BRcc/BBITn/Jcc using auto-update extension core registers, or simply the BLcc instruction.

This mechanism removes many possible hazards that might result from not returning to a faulting instruction that was previously canceled, such as the possibility of TLB thrash or deadlock.

You must not specify a long-immediate source operand in the delay-slot instruction or in the instruction present in the delay slot. When such instructions are encountered, the processor may not fetch the long-immediate value from memory. Therefore, any memory protection mechanism applied to instruction fetch, for example MPU or MMU, is not guaranteed to detect a fetch fault due to the long-immediate operand access. For example, if a delay-slot and its illegal long-immediate data straddles an MPU region or an MMU page boundary, it is possible that only the first MPU region or MMU page is accessed. In such cases an Illegal Instruction Sequence exception may be raised rather than a privilege violation or TLB miss.

When a JL.D, JLcc.D, BL.D or BLcc.D instruction is executed, the size of the following delay-slot instruction is used in the calculation of the BLINK value. Normally, the size of each instruction is determined by its format (2 or 4 bytes) and the presence or absence of a long-immediate data operand.

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). If a delay-slot instruction has a LIMM operand it will raise an Illegal Instruction Sequence exception. 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, including any illegal LIMM operand if present.

However, if the processor does not include any instructions encoded with the same major op-code as the unrecognized delay-slot instruction, the operands of that instruction are not recognized by the processor and it is unaware of the presence of a long-immediate source operand.

In such cases the size of the delay-slot instruction is determined solely by the size of the instruction format. This case could occur, for example, if an APEX instruction is issued when there is no APEX present.

A branch or jump with linkage and a delay slot always observes its delay-slot instruction, to allow it to compute the required BLINK value. However, if an exception is raised when fetching such a delay-slot instruction, then it is not possible for the branch or jump to compute the BLINK value. This is because the delay-slot instruction cannot be trusted and therefore its size cannot be determined.

When such a branch or jump observes its delay-slot instruction, and there is no fetch-related exception for that instruction, the branch instruction completes normally. Subsequently, a non-fetch exception may still be raised on the delay-slot instruction preventing it from completing. If the delay-slot instruction is later restarted, after resolving the condition that raised the exception, it is said to be an orphaned delay slot

instruction. If the faulting delay-slot instruction is later resumed, it must be the same as the one that was observed by the parent branch when it executed.

Whenever a fetch-related exception is raised on a delay-slot instruction, whose parent branch or jump writes to BLINK, the exception is transferred to the parent branch. However, this does not happen to orphaned delay-slot instructions, as their parent branches will have already completed.

When a fetch-related exception is transferred to the parent branch, the ERET register is set to the address of the parent branch, whereas the EFA register is set to the address of the delay-slot instruction. The EFA register points the exception handler to the faulting instruction, and the ERET register defines the address to return to to resume execution from the parent branch."

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

5.4.10 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.

6

Instruction Set Summary

This chapter provides an overview of the ARCV2 ISA. It presents an overview of how the ARCV2 ISA is encoded, explains the assembly syntax conventions used by the remainder of this document, and presents a summary of the instructions found in each functional instruction grouping. Information the encoding of each instruction can be found in [Chapter 7, “32-bit Instruction Formats Reference”](#) and [Chapter 8, “16-bit Instruction Formats Reference”](#), and full details of all aspects of each individual instruction can be found in [Chapter 9, “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 [102](#).

6.1 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.

6.1.1 Top-level Instruction Formats

[Table 6-1](#) 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 6-1 Top-level Formats of the ARCV2 Instruction Set

Format	Description	Examples
F32_BR0	Branch conditionally Branch unconditionally far	Bcc s21 B s25
F32_BR1	Branch-and-link conditionally Branch-and-link unconditionally far Compare-and-branch	BLcc s21 BL s25 BRcc b,c,s9
F32_LD_OFFSET	Load using offset addresses	LD with register + offset addressing
F32_ST_OFFSET	Store using offset addresses	ST with register + offset addressing
F32_GEN4	ARC 32-bit basecase instructions	op a,b,c
F32_EXT5	ARC 32-bit extension instructions	op a,b,c
F32_EXT6	ARC 32-bit extension instructions	op a,b,c
F32_APEX	User 32-bit extension instructions	op a,b,c
F16_COMPACT	Move/Load with h-register	MOV_S g,h / LD_S R0-3,[h,u5]
F16_LD_ADD_SUB	Load/Add/Sub compact	LD_S.AS a,[b,c] / SUB_S a,b,c / ADD_S R0-1,b,u6
F16_LD_ST_1	Load/Store gp-relative Load-indexed	LD R1,[GP,s11] / ST R0,[GP,s11] LDI_S b,[u7]
F16_JLI_EI	Jump-and-link indexed Execute indexed	JLI_S EI_S
F16_LD_ADD_RR	Load/Add register-register	LD_S / LDB_S / LDH_S / ADD_S a,b,c
F16_ADD_IMM	Add/Subtract/Shift immediate	ADD_S / SUB_S / ASL_S / LSR_S c,b,u3
F16_OP_HREG	Move, Compare, Add with one h-register	MOV_S / CMP_S / ADD_S b,h / b,b,h
F16_GEN_OP	General ops and single ops	op_S b,b,c
F16_LD_WORD	Load word with short offset	LD_S c,[b,u7]
F16_LD_BYTE	Load byte with short offset	LDB_S c,[b,u5]
F16_LD_HALF	Load half-word with short offset	LDH_S c,[b,u6]
F16_LDX_HALF	Load signed half-word with short offset	LDH_S.X c,[b,u6]
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]

Table 6-1 Top-level Formats of the ARCV2 Instruction Set

Format	Description	Examples
F16_SH_SUB_BIT	Shift/Subtract/bit ops	op_S b,b,u5
F16_SP_OPS	SP-based Load/Store/Add/Subtract Function prolog, epilog	LD_S / LDB_S / ST_S / STB_S / ADD_S / SUB_S / PUSH_S / POP_S ENTER_S / LEAVE_S
F16_GP_LD_ADD	GP-based Load/Add	LD_S / LDH_S / LDB_S / ADD_S
F16_PCL_LD	PCL-based Load	LD_S b,[PCL,u10]
F16_MV_IMM	Move immediate	MOV_S b,u8
F16_OP_IMM	Add/Compare immediate	ADD_S / CMP_S b,u7
F16_BCC_REG	Branch conditionally on register Z/NZ	BRcc_S b,0,s8
F16_BCC	Branch conditionally	Bcc_S s10/s7
F16_BL	Branch and link unconditionally	BL_S s13
F32_GEN_OP64	ARC 64-bit instructions	op64 a,b,c
F32_VEC1	Vector DSP group 1	(defined in Vector DSP section)
F32_VEC2	Vector DSP group 2	
F32_VEC3	Vector DSP group 3	
F32_VEC4	Vector DSP group 4	
F32_VEC5	Vector DSP group 5	
F32_VEC6	Vector DSP group 6	

6.1.2 Instruction Set Profiles

The ARCV2 instruction set includes the concept of an **instruction set profile**. Each profile allows a small number of top-level formats to encode profile-specific sets of extension instructions that are designed for the application domains of that profile. This acknowledges the fact that a common set of instructions are applicable across all domain, although a unique use of every encoding is too restrictive for all potential use-cases.

The **Baseline Profile** defines a set of common top-level formats that are always included by default in all other profiles.

The **Compact Profile** is designed for general-purpose embedded controller and real-time systems, where both high performance and high code density are required. The Compact Profile extends the set of common formats with several 16-bit formats. This provides a broader set of 16-bit instruction encodings to maximize code density. Other profiles may use those major opcodes in different ways, to encode domain-specific instructions.

6.1.3 Assignment of Instruction Formats to Major Opcodes

Table 6-2 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 6-2 Assignment of top-level instruction formats to major opcodes in each instruction set profile

Major Opcode	Common formats	Profile-specific formats	
	Baseline Profile	Compact Profile	Vector DSP Profile
0x00	F32_BR0		
0x01	F32_BR1		
0x02	F32_LD_OFFSET		
0x03	F32_ST_OFFSET		
0x04	F32_GEN4		
0x05	F32_EXT5		
0x06	F32_EXT6		
0x07	F32_APEX		
0x08	F16_COMPACT		
0x09	F16_LD_ADD_SUB		
0x0A	F16_LD_ST_1		
0x0B		F16_JLI_EI	
0x0C	F16_LD_ADD_RR		
0x0D	F16_ADD_IMM		
0x0E	F16_OP_HREG		
0x0F	F16_GEN_OP		
0x10	F16_LD_WORD		
0x11		F16_LD_BYTE	F32_VEC1
0x12		F16_LD_HALF	F32_VEC2
0x13		F16_LDX_HALF	F32_VEC3
0x14	F16_ST_WORD		
0x15		F16_ST_BYTE	F32_VEC4

Table 6-2 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
0x16		F16_ST_HALF	F32_VEC5
0x17	F16_SH_SUB_BIT		
0x18	F16_SP_OPS		
0x19	F16_GP_LD_ADD		
0x1A	F16_PCL_LD		
0x1B	F16_MV_IMM		
0x1C	F16_OP_IMM		
0x1D	F16_BCC_REG		
0x1E	F16_BCC		
0x1F		F16_BL	F32_VEC6

6.1.5 16-bit Instruction Formats

Table 6-4 summarizes the 16-bit compact instruction formats defined by the ARCV2 instruction set.

Table 6-4 Summary of 16-bit instruction formats

16-bit Formats		Major Opcode					Layout of Instruction Formats										
(a)	(b)	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F16_COMPACT	MOV	0x8					g[2:0]		h[2:0]			g[4:3]		0		h[4:3]	
	LD						U[5]					r[1:0]		U[3:2]			
F16_LD_ADD_SUB	REG_REG	0x9					b[2:0]		c[2:0]		i	0		a[2:0]			
	REG_U6								a[0]	U[5:3]			1				U[2:0]
F16_LD_ST_1	LD_ST_R01	0xA					S[10:5]						i	0		S[4:2]	
	LDI_S						b[2:0]		U[6:3]			1		U[2:0]			
F16_JLI_EI		0xB					i	U[9:0]									
F16_LD_ADD_RR		0xC					b[2:0]		c[2:0]		i[1:0]		a[2:0]				
F16_ADD_IMM		0xD					b[2:0]		c[2:0]		i[1:0]		U[2:0]				
F16_OP_HREG	HREG_REG	0xE					b[2:0]		h[2:0]			i[1:0]		0		h[4:3]	
	HREG_S3						S[2:0]							1			
F16_GEN_OP	DOP	0xF					b[2:0]		c[2:0]		i[4:0] != 00000						
	SOP								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]										

6.1.6 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 369. [Table 6-17](#) on page 270 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 6-5](#) on page 262 and [Table 6-6](#) on page 263 list the notation used for the encoding.

Table 6-5 Key for 32-bit Addressing Modes and Encoding Conventions

Encoding Character	Encoding Field	Syntax
I	I[4:0]	instruction major opcode
i	i[n:0]	instruction sub opcode
A	A[5:0]	destination register
b	B[2:0]	lower bits source/destination register
B	B[5:3]	upper bits source/destination register
C	C[5:0]	source/destination register
Q	Q[4:0]	condition code
u	U[n:0]	unsigned immediate (number is bitfield size)
s	S[n:0]	lower bits signed immediate (number is bitfield size)
S	S[m:n+1]	upper bits signed immediate (number is bitfield size)
T	S[24:21]	upper bits signed immediate (branch unconditionally far)
w	W[5:0]	signed immediate 6-bit store data value
P	P[1:0]	operand format
M	M	conditional instruction operand mode
N	N	<.d> delay slot mode
F	F	Flag Setting
R	R	Reserved

Table 6-5 Key for 32-bit Addressing Modes and Encoding Conventions (Continued)

Encoding Character	Encoding Field	Syntax
D	Di	<.di> direct data cache bypass; this is not applicable to the ARC SEM processors
a	a[1:0]	<.aa> address writeback mode
Z	Z[1:0]	<.zz> data size
X	X	<.x> sign extend
K	K	<.k> store constant operand mode

Table 6-6 Key for 16-bit Addressing Modes and Encoding Conventions

Encoding Character	Encoding Field	Syntax
I	I[4:0]	instruction major opcode
i	i[n:0]	instruction sub-opcode
a	a[2:0]	source/destination register (r0-3,r12-15)
b	b[2:0]	source/destination register (r0-3,r12-15)
c	c[2:0]	source/destination register (r0-3,r12-15)
h	h[2:0]	source/destination register low (r0-r31 excluding r29 and r30)
H	h[4:3]	source/destination register high (r0-r31 excluding r29 and r30)
g	g[2:0]	destination register low (r0-r31 excluding r29 and r30)
G	G[4:3]	destination register high (r0-r31 excluding r29 and r30)
u	u[n:0]	unsigned immediate (number is bitfield size)
s	s[n:0]	signed immediate (number is bitfield size)

6.1.7 Long Immediate Source Operands and Null Destination Operands

Any 6-bit source register field can indicate that long immediate data is used in place of a register value. This is achieved by using the special register number 62 to indicate a long immediate. This can be used multiple times in an instruction. When a source register field is 62, an explicit 32-bit long immediate value follows the instruction word.

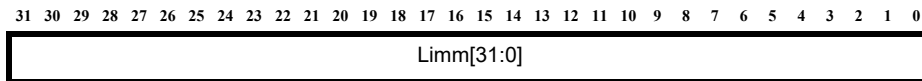
When a destination register field contains 62, the result of the instruction is discarded and no destination register is updated. Any status flag updates still occur according to the set-flags directive (.F) or if flag setting is implicit in the instruction.

If the long immediate indicator is used in both a source and the destination operand, a long immediate value is used as the source operand and the result is discarded as expected.

If the long immediate indicator is used in both source operands, the same long immediate value is used as the value for both source operands. This common long immediate source operand is not duplicated in memory; it is encoded only once and immediately after the instruction as usual.

For a detailed description of the layout in memory for instructions with long immediate data, see to the section on [“Instruction Layout in Memory”](#) on page 120.

Figure 6-1 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)

6.1.8 Condition Code Tests

[Table 6-7](#) lists the codes used for condition code tests.

Table 6-7 Condition Codes

Code Q field	Mnemonic	Condition	Test
0x00	AL, RA	Always	1
0x01	EQ, Z	Zero (Note that EQ and Z are equivalent encodings for condition codes)	Z
0x02	NE, NZ	Non-Zero	/Z
0x03	PL, P	Positive	/N
0x04	MI, N	Negative	N
0x05	CS, C, LO	Carry set, lower than (unsigned)	C
0x06	CC, NC, HS	Carry clear, higher or same (unsigned)	/C
0x07	VS, V	Over-flow set	V
0x08	VC, NV	Over-flow clear	/V
0x09	GT	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)
0x0A	GE	Greater than or equal to (signed)	(N and V) or (/N and /V)
0x0B	LT	Less than (signed)	(N and /V) or (/N and V)
0x0C	LE	Less than or equal to (signed)	Z or (N and /V) or (/N and V)

Table 6-7 Condition Codes (Continued)

Code Q field	Mnemonic	Condition	Test
0x0D	HI	Higher than (unsigned)	/C and /Z
0x0E	LS	Lower than or same (unsigned)	C or Z
0x0F	PNZ	Positive non-zero	/N and /Z

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

- provide additional tests on the internal condition flags or
- test extension status flags from external sources or
- test a combination external and internal flags

If an extension condition code is used which is not implemented, an [Illegal Instruction](#) exception is raised.

6.1.8.1 Condition Code Tests for Floating-Point Instructions

[Table 6-8](#) lists the mapping from the ARCV2 condition codes to the IEEE-754 standard condition codes:

Table 6-8 Condition Codes

IEEE Test	Mnemonic	Condition	Test	Code
	AL, RA	Always	1	0x00
EQ	EQ, Z	Zero	Z	0x01
LT, GT, UN	NE, NZ	Non-Zero	/Z	0x02
EQ, GT, UN	PL, P	Positive	/N	0x03
LT	MI, N	Negative	N	0x04
LT	CS, C, LO	Carry set, lower than (unsigned)	C	0x05
EQ, GT, UN	CC, NC, HS	Carry clear, higher or same (unsigned)	/C	0x06
UN	VS, V	Over-flow set	V	0x07
EQ, LT, GT	VC, NV	Over-flow clear	/V	0x08
GT	GT	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)	0x09
GT, EQ	GE	Greater than or equal to (signed)	(N and V) or (/N and /V)	0x0A
LT, UN	LT	Less than (signed)	(N and /V) or (/N and V)	0x0B
EQ, LT, UN	LE	Less than or equal to (signed)	Z or (N and /V) or (/N and V)	0x0C
GT, UN	HI	Higher than (unsigned)	/C and /Z	0x0D
LT, EQ	LS	Lower than or same (unsigned)	C or Z	0x0E

Table 6-8 Condition Codes

IEEE Test	Mnemonic	Condition	Test	Code
	PNZ	Positive non-zero	/N and /Z	0x0F

**Note**

PNZ does not have an inverse condition.

6.1.9 Encoding Branch and Jump Delay Slot Modes

Table 6-9 lists the codes used for delay slot modes on Branch and Jump instructions.

Table 6-9 Delay Slot Modes

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

6.1.10 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.

6.1.10.1 Assembly Syntax for Static Branch Predictions

Table 6-10 lists the interpretation of the <.T> assembly syntax when encoding static branch predictions within the Y bit of a BRcc or BBITn instruction, according to the sign of the branch displacement.

Table 6-10 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

Table 6-10 Encoding Static Branch Predictions for BRcc, BBIT0, and BBIT1 (Continued)

Instruction Grouping	<.T> Syntax	Prediction	Sign of Displacement	Prediction Mode	Y bit
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 6-10](#) 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 6-10](#) lists these cases in bold font.

6.1.11 Encoding Load / Store Address Writeback Modes

[Table 6-11](#) lists the codes used for address writeback modes in Load and Store instructions.

Table 6-11 Address Writeback Modes

AA bits	Address Mode	Memory Address Used	Register Value Writeback
00	No writeback	Reg + offset	No writeback
01	.A or AW	Reg + offset	Reg + offset Register updated pre memory transaction.
10	.AB	Reg	Reg + offset Register updated post memory transaction.
11	.AS Scaled, no writeback	Reg + (offset << scaling_shift) Note that using the scaled address mode with 8-bit data size (LDB.AS or STB.AS) has undefined behavior and must not be used. scaling_shift is based on the data size mode, ZZ. For more information, see Table 6-14 scaling_shift is defined in Table 6-12 .	No writeback

Table 6-12 Scaling Shift

ZZ Code	ZZ Suffix	scaling_shift
00		2
01	B	undefined
10	H	1
11	D	2

If the destination of a Load instruction is the same as a source register, and the addressing mode defines an update to the source register, the value written to that register is the value returned from memory.

6.1.12 Encoding Load / Store Cache Bypass Mode

[Table 6-13](#) lists the codes used for direct to memory bypass modes in Load and Store instructions.



Note

This section is not applicable to the ARC SEM processors.

Table 6-13 Cache Bypass Modes

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

6.1.13 Encoding Load / Store Data Sizes

[Table 6-14](#) lists the codes used for data size modes in Load and Store instructions.

Table 6-14 Load Store Data Sizes

ZZ Code	ZZ Suffix	Access Mode
00		Word (32-bit), default behavior
01	B	Byte
10	H	Half-word (16-bit)
11	D	Raises an Illegal Instruction exception if LL64_OPTION is disabled, otherwise indicates double-word data.

6.1.14 Encoding Load Data Extension Modes

[Table 6-15](#) lists the codes used to encode extension semantics in Load instructions.

Table 6-15 Load Data Extend Mode

X bit	X Suffix	Extension Semantics
0		If size is not Word(32-bit, ZZ = 00) then data is zero extended.
1	X	If size is not Word (32-bit, ZZ = 00) then data is sign extended

6.1.15 Encoding Store Data Source Mode

Table 6-16 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 6-16 Store Data Source Mode

K bit	K Suffix	Store Data Source
0		The 'c' operand field (Bits 6 through 11) specifies the address of the core register from which the Store Data is sourced.
1		The 'c' operand field contains a signed 6-bit literal value to be stored.

6.1.16 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.

6.1.17 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

6.1.17.1 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.

6.1.17.2 **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.

6.2 **Instruction Syntax Conventions**

The mnemonics of instructions encoded in 16-bit formats normally carry a “_S” suffix to distinguish them from 32-bit formats, and where necessary a 32-bit format may carry a “_L” suffix.

- op implies 32-bit instruction
- op_L indicates 32-bit instruction
- op_S indicates 16-bit instruction

Where the operator format (DOP, SOP and ZOP) is relevant to the syntax or encoding of the instruction. If no suffix is used on the instruction, the implied format is a 32-bit format. Instructions encoded in 16-bit formats have a reduced range of source and target core registers unless indicated otherwise. For an alphabetic list of instructions, see [Table 9-1](#) on page 370. The following notation is used to describe the syntax of instructions.

Table 6-17 **Instruction Syntax Convention**

a	Destination register (reduced range for 16-bit instruction.)
b	Source operand 1 (reduced range for 16-bit instruction) or destination register.
c	Source operand 2 (reduced range for 16-bit instruction) or destination register.
h, g	Full register range for 16-bit instructions; General Purpose register, capable of addressing registers r0-r28 and r31 in some of the 16-bit encoded instructions. When these operands encoded value is 30, the instruction indicates long immediate data rather than r30. And similarly, operand encoded value of 29 is reserved.
A	A 64-bit destination register.
B	A 64-bit source operand register.

Table 6-17 Instruction Syntax Convention (Continued)

C	A 64-bit source operand register.
ACC	Accumulator. In a 64-bit mode, the accumulator comprises a register pair, ACCH and ACCL.
cc	Condition code
<.cc>	Optional condition code
IV	Invalid flag for floating-point unit operations
DZ	Divide-by-zero flag for floating-point unit operations
OF	Overflow flag for floating-point unit operations
UF	Underflow flag for floating-point unit operations
IX	Inexact flag for floating-point unit operations
<.f>	Optional set flags
<.aa>	Optional address writeback
<.d>	Optional delay slot mode
<.di>	Optional direct data cache bypass; This is not applicable to ARC SEM.
<.x>	Optional sign extend
<zz>	Optional data size
u	Unsigned immediate, number indicates field size
s or t	Signed immediate, number indicates field size
limm	Long immediate
.T	Compilers and assembler programmers must use the <.T> syntax to convey the predicted outcome of each BRcc or BBITn instruction to the assembler.

6.3 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.

6.3.1 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.

6.3.2 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

6.4 Arithmetic and Logical Instructions

These instructions perform either dyadic or monadic operators, and take one of the following forms:

$a \leftarrow b \text{ op } c$

$a \leftarrow \text{op } b$

A dyadic instruction applies the operator (op) to the source operands (b and c) and assigns the result to the destination operand (a). The ordering of the operands is important for non-commutative operations, such as, SUB, SBC, BIC, ADD1/2/3, and SUB1/2/3.

A monadic instructions applies the operator (op) to the source operand (b) and assigns the result to the destination operand (a).

All arithmetic and logical instructions can be conditional, or set the flags, or both.

The normal rules concerning the use of r62 to indicate long-immediate source data and/or a null result apply to these instructions (see “[Long Immediate Source Operands and Null Destination Operands](#)” on page 263).

6.4.1 Integer Adder Operations

Table 6-18 summarizes the instructions that perform integer addition, subtraction or comparison operations. These instructions are always supported in all ARCV2 processors.

Table 6-18 Integer addition, subtraction and comparison operations

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

6.4.1.1 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.

6.4.1.2 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.

6.4.1.3 Comparison Instructions

The compare instructions perform 2's complement integer subtraction with the sole purpose of setting the status flags Z, N, C and V, that is, the difference between the source operands is not assigned to a destination register. There is a 16-bit redundant format for the compare instruction, to assist with code density.

There is also a reverse comparison instruction (RCMP), due to the asymmetry of some source operand formats and the non-commutativity of the comparison operator.

6.4.2 Move Operations

[Table 6-18](#) 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 6-19 Move operations

Instruction	Operation	Description
MOV	$a \leftarrow b$	Move source to destination
EXTB	$a \leftarrow \text{extb}(b)$	Zero-extend byte to word
EXTH	$a \leftarrow \text{exth}(b)$	Zero-extend half to word
SEXB	$a \leftarrow \text{sexb}(b)$	Sign-extend byte to word
SEXH, SEXW	$a \leftarrow \text{sexh}(b)$	Sign-extend half to word

6.4.2.1 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 9, "Instruction Set Details"](#).

6.4.2.2 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 9, "Instruction Set Details"](#).

6.4.2.3 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 9, "Instruction Set Details"](#).

6.4.3 Bit-wise Logical Operations

[Table 6-20](#) summarizes the group of instructions that perform bit-wise logical operations. These instructions are always supported in all ARCV2 processors.

Table 6-20 Bit-wise logical operations

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

6.4.4 Bit Operations and Mask Operations

Table 6-21 Bit-mask operations

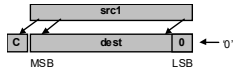
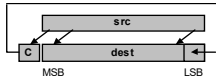
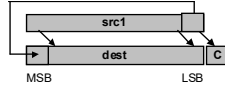
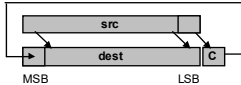
Instruction	Operation	Description
BSET	$a \leftarrow b \text{ or } (1 \ll c)$	bit set
BCLR	$a \leftarrow b \text{ and not } (1 \ll c)$	bit clear
BTST	$b \text{ and } (1 \ll c)$	bit test
BXOR	$a \leftarrow b \text{ xor } (1 \ll c)$	bit xor
BMSK	$a \leftarrow b \text{ and } ((1 \ll (c+1)) - 1)$	bit mask

Table 6-21 Bit-mask operations (Continued)

Instruction	Operation	Description
BMSKN	$a \leftarrow b \text{ and } \sim((1 \ll (c+1)) - 1)$	bit mask inverted

6.4.5 Shift and Rotate Operations

Table 6-22 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
RRC		Rotate right through carry

6.4.5.1 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 writeback to any core register. [Table 6-22](#) on [page 276](#) lists the single bit shift instructions provided as single operand instructions.

Table 6-23 Multi-bit Shift Operations

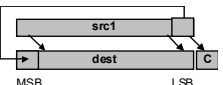
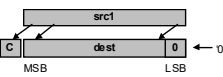
Instruction	Operation	Description
ASR multiple		Multiple arithmetic shift right, sign filled
LSR multiple		Multiple logical shift right, zero filled
ROR Multiple		Multiple rotate right
ASL Multiple		Multiple arithmetic shift left, zero filled
XBFU	$a \leftarrow (b \gg c[4:0]) \text{ and } ((1 \ll (c[9:5]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

Table 6-24 Shift-assist operations

Instruction	Operation	Description
LSL16	$a \leftarrow b \text{ lsl } 16$	Logical shift left 16 places
LSR16	$a \leftarrow b \text{ lsr } 16$	Logical shift right 16 places
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

6.4.6 Selection Operations

Table 6-25 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

6.4.7 Byte-swapping Operations

Table 6-26 Byte-swapping operations

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

6.4.8 Bit-scanning Operations

Table 6-27 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

6.4.9 Relational Comparison Operations

Table 6-28 Relational Comparison Operations

SETcc variant	Operation	Description
SETEQ	$a \leftarrow (b == c)$	Set if equal
SETNE	$a \leftarrow (b != c)$	Set if not equal
SETLT	$a \leftarrow (b < c) \text{ signed}$	Set if less than
SETGE	$a \leftarrow (b \geq c) \text{ signed}$	Set if greater than or equal
SETLO	$a \leftarrow (b < c) \text{ unsigned}$	Set if lower than
SETHS	$a \leftarrow (b \geq c) \text{ unsigned}$	Set if higher or same
SETLE	$a \leftarrow (b \leq c) \text{ signed}$	Set if less than or equal

Table 6-28 Relational Comparison Operations (Continued)

SETcc variant	Operation	Description
SETGT	$a \leftarrow (b > c)$ signed	Set if greater than

6.4.10 Integer Multiply, Multiply-accumulate, and Divide Operations

Table 6-29 Integer Multiply, MAC and Divide Operations

Instruction	Operation	Description
MPY, MPY_S	$a \leftarrow b * c$	32 x 32 signed integer multiply, returning lower 32 bits of product
MPYU	$a \leftarrow b * c$	32 x 32 unsigned integer multiply, returning lower 32 bits of product
MPYM MPYH	$a \leftarrow b * c$	32 x 32 signed integer multiply, returning upper 32 bits of product
MPYMU MPYHU	$a \leftarrow b * c$	32 x 32 unsigned integer multiply, returning upper 32 bits of product
MPYW, MPYW_S	$a \leftarrow b * c$	16 x 16 signed integer multiply, returning 32 bit product
MPYUW, MPYUW_S	$a \leftarrow b * c$	16 x 16 unsigned integer multiply, returning 32 bit product
MPYD	$a \leftarrow b * c$	32 x 32 signed integer multiply, returning 64-bit result
MPYDU	$a \leftarrow b * c$	32 x 32 unsigned integer multiply, returning 64-bit result
MAX	$acc \leftarrow acc + (b * c)$ $a \leftarrow acc\_lo$	32 x 32 signed integer multiply-accumulate, returning lower 32 bits of the accumulated results
MACU	$acc \leftarrow acc + (b * c)$ $a \leftarrow acc\_lo$	32 x 32 unsigned integer multiply-accumulate, returning lower 32 bits of the accumulated results
MACD	$acc \leftarrow acc + (b * c)$ $A \leftarrow acc$	32 x 32 signed integer multiply-accumulate, returning 64 bits of the accumulated results
MACDU	$acc \leftarrow acc + (b * c)$ $A \leftarrow acc$	32 x 32 unsigned integer multiply-accumulate, returning 64 bits of the accumulated results
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

6.4.11 Dual and Quad Integer Multiply / Accumulate Operations

Table 6-30 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
QMPYH	<pre>if (cc) { result = (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3); A = result; acc = result; }</pre>	Quad 16x16 multiply
QMPYHU	<pre>if (cc) { result = (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3); A = result; acc = result; }</pre>	Unsigned QMPYH
DMPYWH	<pre>if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}</pre>	Dual 32x16 multiply
DMPYWHU	<pre>if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}</pre>	Unsigned dual 32x16 multiply
QMACH	<pre>if (cc) { result = acc + (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3); A = result; acc = result; }</pre>	Quad 16x6 MAC

Table 6-30 Dual and Quad Integer Multiply and MAC Operations

Instruction	Operation	Description
QMACHU	<pre>if (cc) { result = acc + (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3); A = result; acc = result; }</pre>	Unsigned quad 16x16 MAC
DMACWH	<pre>if (cc) { result = acc + ((B.w0 * c.h0)+(B.w1 * c.h1)); A = result; acc = result; }</pre>	Dual 32x16 MAC
DMACWHU	<pre>if (cc) { result = acc + ((B.w0 * c.h0)+(B.w1 * c.h1)); A = result; acc = result;} </pre>	Unsigned 32x16 MAC

6.4.12 Integer Vector Operations

Table 6-31 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
VADDSUB2H	<pre>if (cc) { a.h0 = b.h0 + c.h0; a.h1 = b.h1 - c.h1; }</pre>	Vector-add/sub, dual 16-bit
VSUBADD2H	<pre>if (cc) { a.h0 = b.h0 - c.h0; a.h1 = b.h1 + c.h1; }</pre>	Vector-sub/add, dual 16-bit

Table 6-31 Integer Vector ADD, SUB, MPY operations

Instruction	Operation	Description
VMPY2H	<pre> if (cc) { acc.w0 = A.w0 = b.h0 * c.h0; acc.w1 = A.w1 = b.h1 * c.h1; } </pre>	Vector-multiply, dual 16x16
VMPY2HU	<pre> if (cc) { acc.w0 = A.w0 = b.h0 * c.h0; acc.w1 = A.w1 = b.h1 * c.h1; } </pre>	Unsigned vector-multiply, dual 16x16
VADD4H	<pre> if (cc) { A.h0 = B.h0 + C.h0; A.h1 = B.h1 + C.h1; A.h2 = B.h2 + C.h2; A.h3 = B.h3 + C.h3; } </pre>	Vector-add, quad 16-bit
VSUB4H	<pre> if (cc) { A.h0 = B.h0 - C.h0; A.h1 = B.h1 - C.h1; A.h2 = B.h2 - C.h2; A.h3 = B.h3 - C.h3; } </pre>	Vector-sub, quad 16-bit
VADDSUB4H	<pre> if (cc) { A.h0 = B.h0 + C.h0; A.h1 = B.h1 - C.h1; A.h2 = B.h2 + C.h2; A.h3 = B.h3 - C.h3; } </pre>	Vector add/sub, quad 16-bit
VSUBADD4H	<pre> if (cc) { A.h0 = B.h0 - C.h0; A.h1 = B.h1 + C.h1; A.h2 = B.h2 - C.h2; A.h3 = B.h3 + C.h3; } </pre>	Vector-sub/add, quad 16-bit
VADD2	<pre> if (cc) { A.w0 = B.w0 + C.w0; A.w1 = B.w1 + C.w1; } </pre>	Vector-add, dual 32-bit
VSUB2	<pre> if (cc) { A.w0 = B.w0 - C.w0; A.w1 = B.w1 - C.w1; } </pre>	Vector-sub, dual 32-bit

Table 6-31 Integer Vector ADD, SUB, MPY operations

Instruction	Operation	Description
VADDSUB	if (cc) { A.w0 = B.w0 + C.w0; A.w1 = B.w1 - C.w1; }	Vector-add/sub, dual 32-bit
VSUBADD	if (cc) { A.w0 = B.w0 - C.w0; A.w1 = B.w1 + C.w1; }	Vector-sub/add, dual 32-bit

6.5 Memory Access Instructions

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

Table 6-32 Memory Access Instructions

Instruction	Operation	Description
EX	$b \leftarrow \text{mem}[c];$ $\text{mem}[c] \leftarrow b$	Atomic Exchange
LD	Load register-register	See section General Operations Register-register Format on page 328
LDI	$b \leftarrow \text{memory}[\text{LDI\_BASE} + (\text{src1} \ll 2)]$	Load Indexed
LLOCK	$b \leftarrow \text{mem}[c];$ $\text{LF} \leftarrow 1;$	Load locked
SCOND	if $\text{LF mem}[c] \nabla b$	Store conditional
LLOCKD	$B \leftarrow \text{mem}[C];$ $\text{LF} \leftarrow 1;$	Load locked on 64-bit data
SCONDD	if $\text{LF mem}[C] \nabla B$	Store conditional on 64-bit data

The transfer of data to and from memory is accomplished with the load and store commands (LD, ST). These instructions can write the result of the address computation back to the address source register, pre- or post-calculation. This operation is accomplished with the optional address writeback 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.

6.5.1 Load Instructions

Two syntaxes are available, depending on how the address is calculated: register-register and register-offset. The syntax for the load instruction is:

LD<zz><.x><.aa><.di>	a,[b]	(uses ld a,[b,0])
LD<zz><.x><.aa><.di>	a,[b,s9]	
LD<zz><.x><.di>	a,[limm,s9]	(Redundant format, use ld a,[limm])
LD<zz><.x><.di>	a,[limm]	(= ld a,[limm,0])
LD<zz><.x><.aa><.di>	a,[b,c]	
LD<zz><.x><.aa><.di>	a,[b,limm]	
LD<zz><.x><.di>	a,[limm,c]	
LD_S	a, [b, c]	(reduced set of regs)
LDB_S	a, [b, c]	(reduced set of regs)
LDH_S	a, [b, c]	(reduced set of regs)
LD_S	c, [b, u7]	(u7 offset is 32-bit aligned, reduced set of regs)
LDB_S	c, [b, u5]	(reduced set of regs)

LDH_S<.x>	c, [b, u6]	(u6 offset is 16-bit aligned, reduced set of regs)
LD_S	b, [SP, u7]	(u7 offset is 32-bit aligned, reduced set of regs)
LDB_S	b, [SP, u7]	(u7 offset is 32-bit aligned, reduced set of regs)
LD_S	r0, [GP, s11]	(s11 offset is 32-bit aligned, reduced set of regs)
LDB_S	r0, [GP, s9]	(reduced set of regs)
LDH_S	r0, [GP, s10]	(s10 offset is 16-bit aligned, reduced set of regs)
LD_S	b, [PCL, u10]	(u10 offset is 32-bit aligned, reduced set of regs)
LD_S	r, [h, u5]	(r is one of r0-r3, u5 offset is 32-bit aligned, reduced set of regs)
LD_S.AS	a, [b, c]	(reduced set of regs)
LD_S	r1, [GP, s9]	(reduced set of regs)

6.5.2 Store Instructions

6.5.2.1 Store Register with Offset

The following is the syntax for the Store register (**ST 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]	(u7 offset is 32-bit aligned)
STB_S	b, [SP, u7]	(u7 offset is 32-bit aligned)
ST_S	c, [b, u7]	(u7 offset is 32-bit aligned)
STB_S	c, [b, u5]	
STH_S	c, [b, u6]	(u6 offset is 16-bit aligned)

6.5.3 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.

6.5.4 Atomic Memory Operations

An atomic exchange operation, EX, is provided in the ARCV2 architecture as a primitive for multiprocessor synchronization allowing for atomic manipulation of semaphores in memory.

The EX performs an atomic exchange between a memory address and a core register. This operation is atomic in that the memory system ensures that the memory read and memory write cannot be separated by interrupts or by memory accesses from another processor or I/O device.

Two forms of the EX instructions are provided: an uncached form (using the .DI directive) for synchronization between multiple processors, and a cached form for synchronization between processes on a single-processor system. The cached EX is completely handled by the core and does not depend on the AXI subsystem. The uncached EX relies on locked bus transactions that must be supported in the memory subsystem. Note that the locked bus transactions is supported in AXI3 but it is not supported in AXI4.

The ARCV2 architecture also provides special Load-locked and Store-conditional instructions, to enable the implementation of lock-free data structures. These instructions generate exclusive transactions (in AXI this means the command attribute AxLOCK is asserted). The LLOCK instruction loads a value from memory, while at the same time setting an architecturally-invisible lock flag and lock address register. The SCOND instruction atomically checks the status of the lock flag and lock address, and if the check succeeds this instruction writes the result to memory.

Any interrupt or exception clears the lock flag, and any external write to the lock location also clears the lock flag. The lock location is defined to be the 4-byte word location given by the lock address when the LL64_OPTION is disabled, or the 8-byte double-word location given by the lock address when the LL64_OPTION is enabled. All LLOCK, LLOCKD, SCOND, and SCOND 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 SCOND 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.

6.5.5 Prefetch Instructions

The PREFETCH instruction is used to initiate a data cache load without writing to any core register.

The ISA provides three types of prefetch instruction, although some cores may not support all of them. See the relevant core-specific Databook for details:

- PREFETCH: Prefetch data with no intention to write.
- PREFETCHW: Prefetch data with intention to write, that is, the line is allocated in dirty/modified state in the data cache. This option cannot be configured in the ARC EM processor, and therefore this instruction is decoded as an illegal instruction in the ARC EM processor.
- PREALLOC: Allocates a cache line as a preparation for writing the entire line. The line is not fetched from the memory subsystem. This option cannot be configured in the ARC EM processor, and therefore this instruction is decoded as an illegal instruction in the ARC EM processor.

All prefetch-type instructions are encoded as load instructions with a null destination register (r62), in either of the two 32-bit encodings for loads.

The semantics of each load instruction is governed by four orthogonal bit fields within the instruction format:

- <zz> specifies data size: word (00), byte (01), half-word (10) and double-word (11)
- <aa> specifies address update/scaling mode: none (00), pre-incr (01), post-incr (10), scaled (11)
- <d> specifies cache bypass mode: no-bypass (0), bypass (1)
- <x> specifies sign-extension: no sign-extension (0), sign-extension (1)

The <zz> and <aa> fields are orthogonally available also to prefetch-type instructions, allowing data size and addressing mode to be specified.

The <x> and <d> fields specify which particular prefetch-type instruction is encoded, when the destination register is r62 (null destination).

Some combinations of values for <zz>, <aa>, <d>, and <x> are illegal, although this may also depend on whether the instruction is a load or a prefetch-type.

The PREFETCH, PREFETCHW and PREALLOC instructions are encoded using <d> and <x> bits, when the destination register is null (r62), as follows:

Table 6-33 PREFETCH Instruction Encodings

Class	<d>	<x>	Instruction
PREFETCH	0	0	PREFETCH
PREFETCH	0	1	PREALLOC; This instruction is not supported in the ARC EM processor and is decoded as an illegal instruction.
PREFETCH	1	0	PREFETCHW; This instruction is not supported in the ARC EM processor and is decoded as an illegal instruction.
PREFETCH	1	1	Reserved (currently deemed illegal)
Load	-	-	LD, LDB, LDH or LDD, depending on <zz>

The following combinations of load or prefetch-type modifier fields are illegal, and raise an Illegal Instruction exception.

- The <x> and <d> bits together select the prefetch-type opcode.
- The '-' character indicates a don't care case.
- The <aa> and <zz> fields operate identically for both prefetch and load instruction classes. Thus, prefetch-type instructions may have both data-size and address pre/post-increment or scaling, just as load instructions.

All illegal combinations of <d>, <x>, <aa>, and <zz> are shown in [Table 6-34](#). These combinations cover all load and prefetch-type instructions in 32-bit encodings.

Table 6-34 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

6.6 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 6-35 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

6.6.1 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 328](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 329](#), and the [General Operations Register with Signed 12-bit Immediate](#) format on [page 329](#).

6.6.2 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 328](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 329](#), and the [General Operations Register with Signed 12-bit Immediate](#) format on [page 329](#).

6.6.3 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 328](#), the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 329](#), the [General Operations Register with Signed 12-bit Immediate](#) format on [page 329](#), and the [General Operations Conditional Register with Unsigned 6-bit Immediate](#) format on [page 330](#).

6.7 Control Flow Instructions

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

Table 6-36 Control Flow Operations

Instruction	Operation	Description
J	$\text{pc} \leftarrow c$	Unconditional jump
Jcc	if (cc) $\text{pc} \leftarrow c$	Conditional jump
JL	$\text{blink} \leftarrow \text{next\_pc};$ $\text{pc} \leftarrow c$	Jump and link

Table 6-36 Control Flow Operations

Instruction	Operation	Description
JLcc	if (cc) { blink $\leftarrow$ next_pc; pc $\leftarrow$ c }	Jump and link conditionally
JLI_S	blink $\leftarrow$ next_pc; pc $\leftarrow$ JLI_BASE + (u10 $\ll$ 2);	Jump and link indexed
EI_S	BTA $\leftarrow$ next_pc; STATUS32.ES $\leftarrow$ 1 pc $\leftarrow$ EI_BASE + (u10 $\ll$ 2);	Execute indexed
B B_S	pc $\leftarrow$ pcl + offset	Branch unconditionally
Bcc Bcc_S	if (cc) { pc $\leftarrow$ pcl + offset }	Branch conditionally
BLcc	if (cc) { blink $\leftarrow$ next_pc; pc $\leftarrow$ pcl + offset }	Branch-and-link conditionally
BRcc	if (cmp(b,c)) { pc $\leftarrow$ pcl + offset }	Compare-and-branch
BBIT0	if (b[c] == 0) { pc $\leftarrow$ pcl + offset }	Branch if bit is zero
BBIT1	if (b[c] == 1) { pc $\leftarrow$ pcl + offset }	Branch if bit is one
BI	pc $\leftarrow$ next_pc + (c $\ll$ 2)	Branch indexed
BIH	pc $\leftarrow$ next_pc + (c $\ll$ 1)	Branch indexed half-word
LP<cc>	LP_END $\leftarrow$ pcl + s13 LP_START $\leftarrow$ next_pc	Unconditional loop setup (16-bit aligned target address)

Table 6-36 Control Flow Operations

Instruction	Operation	Description
LP<cc>	<pre> if (!cc) { LP_END ← pcl + u7 LP_START ← next_pc } else { pc ← pcl + u7 } </pre>	Conditional loop setup (16-bit aligned target address)
DBNZ	<pre> if (src != 1) { PC ← PCL + offset } src = src - 1 </pre>	Decrement register and branch if resulting value is non-zero.
ENTER_S	Function prolog sequence	Performs a programmable sequence of operations to allocate a stack frame and save the callee-saved registers. This is not strictly a control-flow instruction, but it is grouped most naturally with LEAVE_E, which can perform a jump to BLINK.
LEAVE_S	Function epilog sequence	Performs a programmable sequence of operations to restore callee-saved registers, deallocate a stack frame, and return to the caller.

6.7.1 Branch Instructions

Because of the pipeline in the ARCV2-based processor, a branch instruction does not take effect immediately, but after a delay of one or more cycles, depending on the implementation. The execution of the immediately following instruction after the branch can be enabled to minimize the cost of each branch instruction. Such a following instruction is said to be in the *delay slot*. The modes for specifying the execution of the delay slot instruction are indicated by the optional .d field according to the [Table 6-37](#).

Table 6-37 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 6-7](#) on page 264 lists the condition codes that are available for conditional branch instructions.

6.7.1.1 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

```

6.7.1.2 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 585).

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)

```

6.7.1.3 Branch On Compare or Bit Test

Branch on Compare (BRcc) and Branch on Bit Test (BBIT0, BBIT1) have a branch range of $\pm 256\text{B}$. The branch target address is 16-bit aligned.

The BRcc instruction is similar in execution to a normal compare instruction (CMP) with the addition that a branch occurs if the condition is met. No flags are updated and no ALU result is written back to the register file. A limited set of condition code tests are available for the BRcc instruction as shown in the following table. Table 6-38 lists the additional condition code tests that are available through the effect of reversing the operands.

Table 6-38 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 6-39 Branch on Compare Pseudo Mnemonics, Register-Register

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

Table 6-40 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 6-40 Branch on Compare Pseudo Mnemonics, Register-Immediate

Mnemonic	Condition
BRGT b,u6,s9	Branch if b is greater than u6 (encode as BRGE b,u6+1,s9)
BRLE b,u6,s9	Branch if b is less than or equal u6 (encode as BRLT b,u6+1,s9)
BRHI b,u6,s9	Branch if b is higher than u6 (encode as BRHS b,u6+1,s9)
BRLS b,u6,s9	Branch if b is lower than or same u6 (encode as BRLO b,u6+1,s9)

6.7.2 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 6-41](#).

Table 6-41 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 \(r31\)](#)).

6.7.3 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.

6.7.4 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.

6.8 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 6-42 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
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

6.8.1 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.

6.8.2 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.

6.8.3 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.

6.8.4 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
```

6.8.5 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
```

6.8.6 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.

6.8.7 Synchronization Instructions

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

6.9 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.

6.9.1 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

6.9.2 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

6.9.3 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

6.10 Floating-Point Instructions

Table 6-43 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
0x09	FCVT32_64	0	Convert from 32-bit data formats (single-precision float, signed, or unsigned integer) to 64-bit data formats
0x2F	FDSQRT	0	Double-precision floating-point square root
0x2F	FSSQRT	0	Single-precision floating-point square root
0x30	FDMUL	0	Double-precision floating-point multiplication
0x31	FDADD	0	Double-precision floating-point addition
0x32	FDSUB	0	Double-precision floating-point subtraction
0x33	FDCMP	1	Double-precision floating-point comparison
0x34	FDCMPF	1	Double-precision floating-point comparison operation with IEEE 754 flag generation.
0x35	FDMADD	0	Double-precision floating-point fusedMultiplyAdd.
0x36	FDMSUB		Double-precision floating-point fusedMultiplySubtract.

Table 6-43 Floating-Point Unit Instructions, 0x06

Sub-opcode B field (6 bits)	Instruction	F Bit	Description
0x37	FDDIV	0	Double-precision floating-point division
0x38	FCVT64	0	Convert between two 64-bit data formats
0x39	FCVT64_32	0	Convert from 64-bit data formats to 32-bit data formats
BXOR r0, r2, 0x1F	FDNEG		Double-precision floating-point negate operation.
BCLR r1, r2, 0x1F	FSABS		Single-precision floating-point absolute operation
BXOR r1, r2, 0x1F	FSNEG		Single-precision floating-point negate operation
BCLR r0, r2, 0x1F	FDABS		Double-precision floating-point absolute operation.

6.10.0.1 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

6.10.0.2 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

6.11 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 6-44 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 6-44 Instruction Mnemonics

L	Least significant
M	Most significant

6.11.1 Basic Saturating Arithmetic Operations

Table 6-45 Basic Saturating Arithmetic Operations

Instruction	Type	Major	Minor	Description
SATF	SOP	0x05	0x1A	Saturate value based on flag
SATH	SOP	0x05	0x02	Saturate a 32-bit value to a 16-bit value
RNDH	SOP	0x05	0x03	Round and saturate a 32-bit value to a 16-bit value
ABSSH	SOP	0x05	0x04	Compute the absolute value of a 16-bit element and saturate the result
ABSS	SOP	0x05	0x05	Compute the absolute value of a 32-bit operand and saturate the result
NEGSH	SOP	0x05	0x06	Negate the lower 16 bits of a word and saturate the result
NEGS	SOP	0x05	0x07	Negate a 32-bit word and saturate the result
ADDS	DOP	0x05	0x06	32-bit addition, and result is saturated
SUBS	DOP	0x05	0x07	32-bit subtraction, and result is saturated
ASLS	DOP	0x05	0x0A	Perform an arithmetic shift left operation on an operand. The shift amount specified in the second operand. Store the saturated result in the destination
ASRS	DOP	0x05	0x0B	Perform an arithmetic shift right operation on an operand. The shift amount specified in the second operand. Store the saturated result in the destination

Table 6-45 Basic Saturating Arithmetic Operations

Instruction	Type	Major	Minor	Description
ASRSR	DOP	0x05	0x0C	Perform an arithmetic shift right operation on an operand. The shift amount specified in the second operand. Store the rounded saturated result in the destination
ADCS	DOP	0x05	0x26	32-bit add with carry, saturating the result
SBCS	DOP	0x05	0x27	32-bit subtract with borrow, saturating the result

6.11.2 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 112. Short immediates are replicated for each vector lane.

Table 6-46 Vector Unpacking Operations

Instruction	Type	Major	Minor	F bit	Description
VEXT2BHLF	SOP	0x05	0x20	0	Compose a vector with two Q15 elements by expanding the lower two Q7 bytes of an operand.
VEXT2BHMF	SOP	0x05	0x21	0	Compose a vector with two Q15 elements created by expanding the higher two Q7 bytes of an operand.
VREP2HL	SOP	0x05	0x22	0	Replicate lower 16-bits into the upper 16-bits
VREP2HM	SOP	0x05	0x23	0	Replicate higher 16-bits to lower 16-bits
VEXT2BHL	SOP	0x05	0x24	0	Vector extend the lower two bytes to the higher two bytes
VEXT2BHM	SOP	0x05	0x25	0	Vector extend the higher two bytes to the lower two bytes

Table 6-46 Vector Unpacking Operations

Instruction	Type	Major	Minor	F bit	Description
VSEXT2BHL	SOP	0x05	0x26	0	Vector sign extend the lower two bytes to the higher two bytes
VSEXT2BHM	SOP	0x05	0x27	0	Vector sign extend the higher two bytes to the lower two bytes
VPACK2HL	DOP	0x05	0x29	0	Compose the destination operand from the lower 16-bits of the source operands.
VPACK2HM	DOP	0x05	0x29	1	Compose the destination operand from the higher 16-bits of the source operands.
VPACK2HBL	SOP	0x05	0x1C	0	Pack the LSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the lower 16 bits of the destination.
VPACK2HBLF	SOP	0x05	0x1D	0	Pack the MSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the lower 16 bits of the destination.
VPACK2HBM	SOP	0x05	0x1E	0	Pack the LSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the higher 16 bits of the destination.
VPACK2HBMF	SOP	0x05	0x1F	0	Pack the MSBs of a 2x16 bit vectors into a 2x8 bit vector. Result is stored in the higher 16 bits of the destination.
VALGN2H	DOP	0x05	0x0D	0	Two-way 16-bit vector align

6.11.3 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 112. Short immediates are replicated for each vector lane.

Table 6-47 Vector ALU Operations

Instruction	Type	Major	Minor	F bit	Description
VABS2H	SOP	0x05	0x28	0	Two way 16 bit vector absolute
VABSS2H	SOP	0x05	0x29	0	Two way 16 bit vector saturating absolute
VNEG2H	SOP	0x05	0x2A	0	Two way 16 bit vector negative
VNEGS2H	SOP	0x05	0x2B	0	Two way 16 bit vector saturating negative
VNORM2H	SOP	0x05	0x2C	0	Two way 16 bit vector normalization
VADD2H	DOP	0x05	0x14	0	Two way vector add 16 bits
VADDS2H	DOP	0x05	0x14	1	Two way vector saturating add 16 bits
VSUB2H	DOP	0x05	0x15	0	Two way vector subtract 16 bits
VSUB4H	DOP	0x05	0x15	1	Two way vector saturating subtract 16 bits
VADDSUB2H	DOP	0x05	0x16	0	Two way vector add subtract
VADDSUBS2H	DOP	0x05	0x16	1	Two way vector saturating add subtract
VSUBADD2H	DOP	0x05	0x17	0	Two way vector subtract add
VSUBADDS2H	DOP	0x05	0x17	1	Two way vector saturating subtract add
VASL2H	DOP	0x05	0x21	0	Two way arithmetic shift left 16 bits
VASLS2H	DOP	0x05	0x21	1	Two way saturating arithmetic shift left 16 bits
VASR2H	DOP	0x05	0x22	0	Two way arithmetic shift right 16 bits
VASRS2H	DOP	0x05	0x22	1	Two way arithmetic saturating shift right 16 bits
VLSR2H	DOP	0x05	0x23	0	Two way vector logic shift right 16 bits
VASRSR2H	DOP	0x05	0x23	1	Two way arithmetic saturating and rounding shift right 16 bits

Table 6-47 Vector ALU Operations

Instruction	Type	Major	Minor	F bit	Description
VADD4B	DOP	0x05	0x24	0	Four way add
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

6.11.4 Accumulator Operations

Table 6-48 Accumulator Operations

Instruction	Type	Major	Minor	F bit	Description
ASLACC	ZOP	0x05	0x00	0	Arithmetic shift left accumulator
ASLSACC	ZOP	0x05	0x01	0	Saturating arithmetic shift left accumulator
FLAGACC	ZOP	0x05	0x04	1	Copy accumulator status flags to STATUS32
GETACC	SOP	0x05	0x18	0	Read accumulator window
NORMACC	SOP	0x05	0x19	0	Normalize the accumulator
SETACC	DOP	0x05	0x0D	1	Set accumulator

6.11.5 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 112. Short immediates are replicated for each vector lane.

Table 6-49 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

Table 6-49 Vector 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
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
VMPY2HWF	DOP	0x05	0x20	0	Two way vector 16 bit saturating fractional multiplication; two way 32 bit

6.11.6 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 112. Short immediates are replicated for each vector lane.

Table 6-50 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 writeback
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

6.11.7 32x32 and 16x16 MAC Operations

Table 6-51 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

Table 6-51 32x32 and 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MPYMU MPYHU	DOP	0x04	0x1C	F	32 bit unsigned integer multiplication
MPYU	DOP	0x04	0x1D	F	32 bit unsigned integer multiplication
MPYW	DOP	0x04	0x1E	F	16 bit signed integer multiplication
MPYUW	DOP	0x04	0x1F	F	16 bit unsigned integer multiplication
MAC	DOP	0x05	0x0E	F	32 bit signed integer multiplication and accumulation
MACU	DOP	0x05	0x0F	F	32 bit unsigned integer multiplication and accumulation
MPYD	DOP	0x05	0x18	F	32 bit signed integer multiplication; 64-bit result
MPYDU	DOP	0x05	0x19	F	32 bit unsigned integer multiplication; 64-bit result
MACD	DOP	0x05	0x1A	F	32 bit signed integer multiplication and accumulation; 64-bit result
MACDU	DOP	0x05	0x1B	F	32 bit unsigned integer multiplication and accumulation; 64-bit result
MPYF	DOP	0x06	0x0A	F	32 bit signed fractional saturating multiplication
MPYFR	DOP	0x06	0x0B	F	32 bit signed fractional saturating an rounding multiplication
MACF	DOP	0x06	0x0C	F	32 bit signed fractional saturating multiplication and accumulation
MACFR	DOP	0x06	0x0D	F	32 bit signed fractional saturating an rounding multiplication and accumulation
MSUBF	DOP	0x06	0x0E	F	32 bit fractional saturating subtraction
MSUBFR	DOP	0x06	0x0F	F	32 bit fractional saturating and rounding subtraction
MPYDF	DOP	0x06	0x12	F	32 bit signed fractional saturating multiplication; 64-bit result

Table 6-51 32x32 and 16x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MACDF	DOP	0x06	0x13	F	32 bit signed fractional saturating multiplication and accumulation; 64-bit result
MSUBDF	DOP	0x06	0x15	F	32 bit signed fractional saturating subtraction; 64-bit result
MPYW_S	DOP	0x0F	0x09	F	16 bit signed integer multiplication
MPYUW_S	DOP	0x0F	0x0A	F	16 bit unsigned integer multiplication
MPY_S	DOP	0x0F	0x0C	F	32 bit signed integer multiplication

6.11.8 Dual 16x8 MAC Operations



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in “[Expansion of Literals in Vector Instructions](#)” on page 112. Short immediates are replicated for each vector lane.

Table 6-52 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

6.11.9 32x16 MAC Operations

Table 6-53 32x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MPYWHL	DOP	0x06	0x1C	F	32x16 bit signed multiplication, lower signed 16 bits

Table 6-53 32x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
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
MPYWHFM	DOP	0x06	0x20	F	32x16 bit signed fractional multiplication, higher signed 16 bits
MPYWHFMR	DOP	0x06	0x21	F	32x16 bit signed fractional multiplication with rounding, higher signed 16 bits
MACWHFM	DOP	0x06	0x22	F	32x16 bit signed fractional multiplication and accumulation, higher signed 16 bits
MACWHFMR	DOP	0x06	0x23	F	32x16 bit signed fractional multiplication and accumulation with rounding, higher signed 16 bits
MPYWHFL	DOP	0x06	0x24	F	32x16 bit signed fractional multiplication, lower signed 16 bits
MPYWHFLR	DOP	0x06	0x25	F	32x16 bit signed fractional multiplication with rounding, lower signed 16 bits
MACWHFL	DOP	0x06	0x26	F	Signed fractional multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. The result is saturated and rounded.
MACWHFLR	DOP	0x06	0x27	F	Signed 32x16 fractional multiplication and accumulation with rounding.
MACWHKL	DOP	0x06	0x28	F	Signed 32x16 integer multiplication and accumulation and shift-right the lower bits
MACWHKUL	DOP	0x06	0x29	F	Unsigned 32x16 integer multiplication and accumulation and shift-right the lower bits

Table 6-53 32x16 MAC Operations

Instruction	Type	Major	Minor	F bit	Description
MPYWHKL	DOP	0x06	0x2A	F	Signed integer multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator.
MPYWHKUL	DOP	0x06	0x2B	F	Unsigned integer multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator.
MSUBWHFL	DOP	0x06	0x14	F	Signed fractional multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. Subtract the product from the wide accumulator.
MSUBWHFLR	DOP	0x06	0x1A	F	Signed fractional multiplication of 32 bits of the b operand and the least-significant 16 bits of the c operand. Subtract the product from the wide accumulator. Return the rounded and saturated result.
MSUBWHFM	DOP	0x06	0x2C	F	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand.
MSUBWHFMR	DOP	0x06	0x2D	F	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand; with rounded result

6.11.10 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 112. Short immediates are replicated for each vector lane.

Table 6-54 Complex Operations

Instruction	Type	Major	Minor	F bit	Description
CBFLYHF1R	SOP	0x06	0x19	0	16 + 16 bit complex FFT butterfly, second half;
CMPYHNFR	DOP	0x06	0x00	1	16 + 16 bit complex signed fractional multiplication with rounding, no fractional shift
CMPYHFR	DOP	0x06	0x01	1	16 + 16 bit complex signed fractional multiplication with rounding
CMPYHFR	DOP	0x06	0x02	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding
CMPYCHFR	DOP	0x06	0x05	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding
CMACHNFR	DOP	0x06	0x06	1	16 + 16 bit complex conjugated signed fractional multiplication with rounding with no fractional shift
CMACHFR	DOP	0x06	0x07	1	16 + 16 bit complex signed fractional multiplication and accumulation with rounding
CMACCHNFR	DOP	0x06	0x08	1	16 + 16 bit complex conjugated signed fractional multiplication and accumulation with rounding with no fractional shift
CMACCHFR	DOP	0x06	0x09	1	16 + 16 bit complex conjugated signed fractional multiplication and accumulation with rounding
CMPYHFMR	DOP	0x06	0x1B	0	16 + 16 bit complex signed fractional multiplication with rounding
CBFLYHF0R	DOP	0x06	0x1B	1	16 + 16 bit complex FFT butterfly, first half

6.11.11 Vector 64-Bit Operations

Table 6-55 Vector 64-Bit Operations

Instruction	Type	Major	Minor	F bit	Description
QMPYH	DOP	0x5	0x30		Quad 16x16 multiplication
QMPYHU	DOP	0x5	0x31		Quad unsigned 16x16 multiply
DMPYWH	DOP	0x5	0x32		Sum of dual 32x16 multiplication
DMPYWHU	DOP	0x5	0x33		Sum of dual unsigned 32x16 multiplication
QMACH	DOP	0x5	0x34		Quad 16x16 multiply and accumulate
QMACHU	DOP	0x5	0x35		Quad 16x16 unsigned multiply and accumulate
DMACWH	DOP	0x5	0x36		Dual 32x16 multiply and accumulate.
DMACWHU	DOP	0x5	0x37		Dual unsigned 32x16 multiply and accumulate
QMPYHF	DOP	0x6	0x3C		Quad 16x16 fractional multiplication
DMPYHWF	DOP	0x6	0x3D		Sum of dual 16x16 signed fractional multiplication
QMACHF	DOP	0x6	0x3E		Quad 16x16 signed fractional multiply and accumulate; result is saturated
DMACWHF	DOP	0x6	0x3F		Dual 32x16 fractional multiply and accumulate
VADD4H	DOP	0x5	0x38	0	Quad 16-bit SIMD addition
VADDS4H	DOP	0x5	0x38	1	Four-way 16-bit vector addition.
VSUB4H	DOP	0x5	0x39	0	Quad 16-bit SIMD subtraction
VSUBS4H	DOP	0x5	0x39	1	Four-way 16-bit vector subtraction
VADDSUB4H	DOP	0x5	0x3A	0	Quad 16-bit SIMD add and subtract.
VADDSUBS4H	DOP	0x5	0x3A	1	Four-way 16-bit vector addition and subtraction.

Table 6-55 Vector 64-Bit Operations

Instruction	Type	Major	Minor	F bit	Description
VSUBADD4H	DOP	0x5	0x3B	0	Quad16-bit SIMD subtract and add
VSUBADDS4H	DOP	0x5	0x3B	1	Four-way 16-bit vector subtraction and addition
VADD2	DOP	0x5	0x3C	0	Dual 32-bit addition
VADDS2	DOP	0x5	0x3C	1	Two-way 32-bit vector addition.
VSUB2	DOP	0x5	0x3D	0	Dual 32-bit SIMD subtract.
VSUBS2	DOP	0x5	0x3D	1	Two-way 32-bit vector subtraction.
VADDSUB	DOP	0x5	0x3E	0	Dual 32-bit SIMD add and subtract.
VADDSUBS	DOP	0x5	0x3E	1	Two-way 32-bit vector addition and subtraction
VSUBADD	DOP	0x5	0x3F	0	Dual 32-bit SIMD subtract and add.
VSUBADDS	DOP	0x5	0x3F	1	Two-way 32-bit vector subtraction and addition

6.11.12 Divide and Square-Root Instructions



Note

This group of instructions operate on vectors and follow the vector literal expansion rules as defined in [“Expansion of Literals in Vector Instructions”](#) on page 112. Short immediates are replicated for each vector lane.

Table 6-56 Divide and Square-Root Operations

Instruction	Type	Major	Minor	F bit	Description
DIV	DOP	0x05	0x04	F	2's Complement Integer Divide
DIVU	DOP	0x05	0x05	F	Unsigned integer division
REM	DOP	0x05	0x08	F	2's complement integer Remainder
REMU	DOP	0x05	0x09	F	Unsigned integer Remainder
DIVF	DOP	0x06	0x10	F	Signed 32-bit fractional division

Table 6-56 Divide and Square-Root Operations

Instruction	Type	Major	Minor	F bit	Description
SQRT	SOP	0x05	0x30	F	Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result
SQRTF	SOP	0x05	0x31	F	Compute the square root of an unsigned Q31 fraction, yielding a Q31 fractional result.

6.11.13 Address Generation Instructions

Table 6-57 Address Generation Instructions

Instruction	Type	Major	Minor	F bit	Description
MODIF	ZOP	0x05	0x05	0	Apply modifier from immediate
MODAPP	XOP	0x05	0x03E	0	Apply modifier from register

7

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 255 lists and summarizes the 32-bit formats. In particular, [Table 6-4](#) on page 261 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 6-17](#) on page 270, and section [“Encoding Notation”](#) on page 262 describes the encoding notation used when describing instruction encodings in this chapter.

This chapter describes the following 32-bit ARCV2 instruction formats:

- [Branch Format \[F32_BR0\]](#)
- [BRcc, BBITn and BL Format \[F32_BR1\]](#)
- [Load Register with Offset Format \[F32_LD_OFFSET\]](#)
- [Store Register with Offset Format \[F32_ST_OFFSET\]](#)
- [General Operations Format \[F32_GEN4\]](#)
- [APEX Extension Instruction Format \[F32_APEX\]](#)

Each section provides details of the sub-formats, and describes how instructions and their operands are encoded in those sub-formats.

For a summary of the instructions discussed in this chapter, and an explanation of the functions they perform, see [Chapter 6, “Instruction Set Summary”](#).

For full details on the available encodings and syntax for each instruction, see to the relevant page for each instructions in [Chapter 9, “Instruction Set Details”](#).

7.1 Branch Format [F32_BR0]

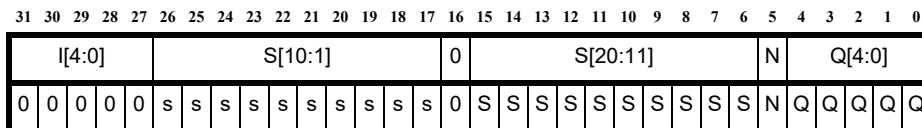
The F32_BR0 format is used to encode branch instructions with large displacements. There are two sub-formats, COND and UCOND_FAR. The COND format is used to encode conditional branch instructions, with an optional delay slot, and the UCOND_FAR format is used to encode unconditional branches with maximum 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.

7.1.1 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 6-7](#) on page 264. For information about the encoding of delay-slot modes, see [Table 6-9](#) on page 266.

Figure 7-1 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.

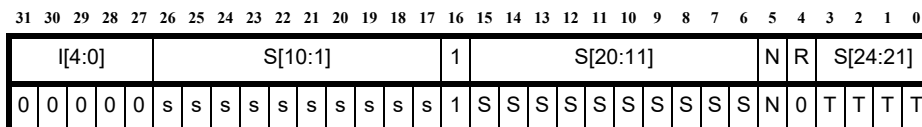
Syntax:

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

7.1.2 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 6-9](#) on page 266 for information on delay slot mode encoding.

Figure 7-2 Branch Unconditional Far Format



Syntax:

B<.d> s25 (unconditional branch far)

7.2 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 6-9](#) on page 266.

These instructions also provide a way to encode a static branch prediction. This is a hint to hardware that, in the absence of a dynamic prediction, the branch is statistically more likely to go one way than the other. Not all implementations of the ARCV2 ISA exploit these hints, but compilers should attempt to provide the most appropriate hint to each BRcc or BBITn instruction, where possible. The encoding of static prediction hints is part of the instruction encoding for this group of instructions, and can be seen in “[Static Branch Prediction Mode](#)” on page 322

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).

7.2.1 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 7-3 Branch on Compare Register-register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			S[7:1]							1	S8	B[5:3]			C[5:0]					N	0	i[3:0]					
0	0	0	0	1	b	b	b	s	s	s	s	s	s	s	1	S	B	B	B	C	C	C	C	C	C	N	0	i	i	i	i

Syntax:

BRcc<.d><.T> b,c,s9 (branch if reg-reg compare is true, swap regs if inverse condition required)

BRcc<.T> b,limm,s9 (branch if reg-limm compare is true)

BRcc<.T> limm,c,s9 (branch if limm-reg compare is true)

BBIT0<.d><.T> b,c,s9 (branch if bit c in reg b is clear)

BBIT0<.T> b,limm,s9 (branch if reg-limm compare is true)

BBIT0<.T> limm,c,s9 (branch if limm-reg compare is true)

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

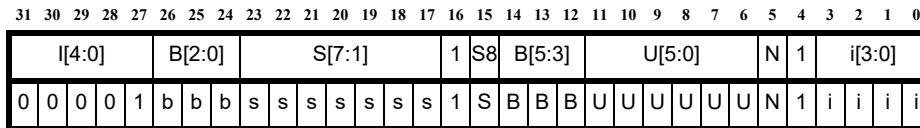
BBIT1<.T> b,limm,s9 (branch if reg-limm compare is true)

BBIT1<.T> limm,c,s9 (branch if limm-reg compare is true)

7.2.2 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 7-4 Branch on Compare Register-Immediate Format



Syntax:

BRcc<.d><.T> b,u6,s9 (branch if reg-immediate compare is true, use "immediate+1" if a missing condition is required)

BBIT0<.d><.T> b,u6,s9 (branch if bit u6 in reg b is clear)

BBIT1<.d><.T> b,u6,s9 (branch if bit u6 in reg b is set)

7.2.2.1 Static Branch Prediction Mode

Table 6-10 on page 266 lists the interpretation of the <.T> syntax for statically-predicted branch instructions. For best performance, compilers or assembly programmers must ensure that the static branch prediction is set according to the most probable outcome of each branch, based either on profiling information or deductive analysis of the program. Each statically-predictable instruction has a prediction mode that is either BTFN or FTBN. These combine with the sign of the branch displacement to give a prediction of taken or not-taken. The BTFN mode predicts a branch to be “backwards taken, forwards not-taken”. Conversely, a FTBN mode predicts a branch to be “forwards taken, backwards not-taken”. The BTFN mode should normally be the default mode generated by a compiler in the absence of profiling information or static deductive reasoning about the branch direction. A BTFN prediction mode typically gives acceptable results on the majority of branches.

7.2.2.2 Encoding of BRcc, BBIT0, and BBIT1 Instructions

Table 7-1 Branch on Compare/Bit Test Register-Register

Sub-opcode i field (4 bits)	Instruction	Operation	Static Branch Prediction	Branch Condition Tested
0x00	BREQ	src1 – src2	BTFN	src1 is equal to src2
0x01	BRNE	src1 – src2	BTFN	src1 and src2 are not equal
0x02	BRLT	src1 – src2	BTFN	src1 is less than src2 (signed)
0x03	BRGE	src1 – src2	BTFN	src1 is greater or equal to src2 (signed)
0x04	BRLO	src1 – src2	BTFN	src1 is lower than src2 (unsigned)

Table 7-1 Branch on Compare/Bit Test Register-Register (Continued)

Sub-opcode i field (4 bits)	Instruction	Operation	Static Branch Prediction	Branch Condition Tested
0x05	BRHS	src1 – src2	BTFN	src1 higher than or same as src2 (unsigned)
0x06	BBIT0	(src1 and 2 ^{src2}) == 0	FTBN	bit src2 in src1 is clear
0x07	BBIT1	(src1 and 2 ^{src2}) == 0	FTBN	bit src2 in src1 is set
0x08	BREQ	src1 – src2	FTBN	src1 is equal to src2
0x09	BRNE	src1 – src2	FTBN	src1 and src2 are not equal
0x0A	BRLT	src1 – src2	FTBN	src1 is less than src2 (signed)
0x0B	BRGE	src1 – src2	FTBN	src1 is greater or equal to src2 (signed)
0x0C	BRLO	src1 – src2	FTBN	src1 is lower than src2 (unsigned)
0x0D	BRHS	src1 – src2	FTBN	src1 higher than or same as src2 (unsigned)
0x0E	BBIT0	(src1 and 2 ^{src2}) == 0	BTFN	bit src2 in src1 is clear
0x0F	BBIT1	(src1 and 2 ^{src2}) == 0	BTFN	bit src2 in src1 is set

7.2.3 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.

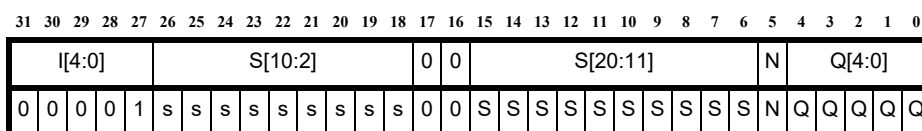
All function entry points are aligned to a 4-byte boundary, requiring that the target address is also 4-byte aligned. Thus the least-significant two bits of the branch offset can be assumed to be zero, and are omitted from the instruction encoding.

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 6-9](#).

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.

7.2.3.1 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 6-7](#) on page 264.

Figure 7-5 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.

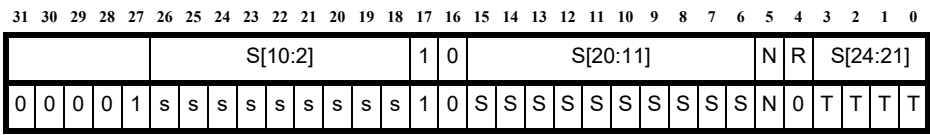
Syntax:

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

7.2.3.2 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 7-6 Branch and Link Unconditional Far Format



The reserved field, R, is ignored.

Syntax:

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

7.3 Load Register with Offset Format [F32_LD_OFFSET]

For information about encoding the Load instruction, see [Table 6-11](#) on page 267, [Table 6-12](#) on page 268, [Table 6-14](#) on page 268, and [Table 6-15](#) on page 269.

Figure 7-7 Load Register with Offset Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			S[7:0]								S8	B[5:3]			Di	a	a	Z	Z	X	A[5:0]						
0	0	0	1	0	b	b	b	s	s	s	s	s	s	s	s	S	B	B	B	D	a	a	Z	Z	X	A	A	A	A	A	A

The program counter (PCL) is not permitted to be the destination of a load instruction. Values 0x3D and 0x3F in the destination register field, A, raise an **Illegal Instruction** exception.

The loop count register (LP_COUNT) is not permitted to be the destination of a load instruction, indicated by A=0x3C. Any use of LP_COUNT as the destination register raises an **Illegal Instruction** exception.

A value of 0x3 in the data size mode field, ZZ, raises an **Illegal Instruction** exception when LL64_OPTION is disabled.

The sign extension field, X, must not be set when the load is of 32-bit word data ZZ=0x0. This combination raises an **Illegal Instruction** exception.

Using incrementing addressing modes, with a long immediate value as the base register, is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, raises an **Illegal Instruction** exception.

Syntax:

LD<zz><.x><.aa><.di> a,[b,s9]

LD<zz><.x><.di> a,[limm,s9] (*use ld a,[limm]*)

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

7.4 Store Register with Offset Format [F32_ST_OFFSET]

For information about encoding the Store instruction, see [Table 6-11](#) on page 267, [Table 6-14](#) on page 268, and [Table 6-14](#) on page 268.

Figure 7-8 Store Register with Offset Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			S[7:0]								S8	B[5:3]			C[5:0]					Di	A	A	Z	Z	K		
0	0	0	1	1	b	b	b	s	s	s	s	s	s	s	s	S	B	B	B	C	C	C	C	C	C	D	a	a	Z	Z	0

Figure 7-9 Store Immediate with Offset Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			S[7:0]								S8	B[5:3]			W[5:0]					Di	A	A	Z	Z	K		
0	0	0	1	1	b	b	b	s	s	s	s	s	s	s	S	B	B	B	w	w	w	w	w	w	w	D	a	a	Z	Z	1

A value of 0x3 in the data size mode field, ZZ, raises an [Illegal Instruction](#) exception when LL64_OPTION is disabled.

Using incrementing addressing modes, with a long immediate value as the base register, is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, raises an [Illegal Instruction](#) exception.

Syntax:

ST<zz><.aa><.di> c,[b,s9]

ST<zz><.di> c,[limm] (= st c,[limm,0])

ST<zz><.aa><.di> limm,[b,s9]

7.5 General Operations Format [F32_GEN4]

The ARCV2 instruction set defines a collection of hierarchical formats that are used to encode the majority of instructions that are neither branches nor load-store operations. These typically define a monadic or dyadic operator, which is applied 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.

7.5.1 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 7-2 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 7-2 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]															
		REG_U6	>													01														U[5:0]																			
		REG_S12	>													10														S[5:0]												S[11:6]							
		COND_REG	>													C[5:0]														0	Q[4:0]																		
		COND_U6	>													U[5:0]														1																			
	SOP	REG_REG	>	F32_GEN4								B[2:0]				00	1 0 1 1 1 1								F	B[5:3]				C[5:0]				i[5:0] (except 111111)															
		REG_U6	>													01														U[5:0]																			
	ZOP	REG_REG	>	F32_GEN4								i[2:0]				00	1 0 1 1 1 1								F	i[5:3]				C[5:0]				1 1 1 1 1 1															
		REG_U6	>													01														U[5:0]																			

7.5.2 Operand Format Indicators

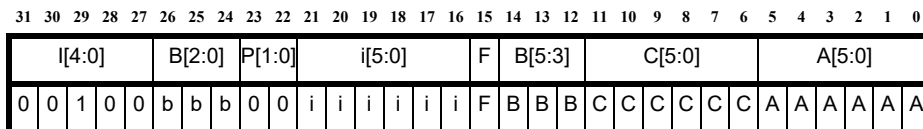
The General Operations formats provide five operand sub-formats, encoded with the P[1:0] and M fields, located in instruction bits [23:22] and [5] respectively, as shown in [Table 7-3](#).

Table 7-3 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

7.5.3 Syntax and Encoding of Instructions in General Operations Formats

7.5.3.1 General Operations Register-register Format

Figure 7-10 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 329](#))

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)

7.5.3.2 General Operations Register with Unsigned 6-bit Immediate

Figure 7-11 General Operations Register with Unsigned 16-bit Immediate Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			U[5:0]					A[5:0]									
0	0	1	0	0	b	b	b	0	1	i	i	i	i	i	i	F	B	B	B	u	u	u	u	u	u	A	A	A	A	A	A

Syntax:

dop<.f> a,b,u6

dop<.f> a,limm,u6 (If B = 62, not useful format)

dop<.f> 0,b,u6 (If A = 62)

dop<.f> 0,limm,u6 (If A = B = 62, not useful format)

sop<.f> b,u6

sop<.f> 0,u6 (If B = 62)

zop<.f> u6

7.5.3.3 General Operations Register with Signed 12-bit Immediate

Figure 7-12 General Operations Register with Signed 12-bit Immediate Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			S[5:0]					S[11:6]									
0	0	1	0	0	b	b	b	1	0	i	i	i	i	i	i	F	B	B	B	s	s	s	s	s	s	S	S	S	S	S	S

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

Syntax:

dop<.f> b,b,s12

dop<.f> 0,limm,s12 (If B = 62, not a useful format)

7.5.3.4 General Operations Conditional Register

Figure 7-13 General Operations Conditional Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			P[1:0]		i[5:0]				F	B[5:3]			C[5:0]					M	Q[4:0]								

Figure 7-13 General Operations Conditional Register Format

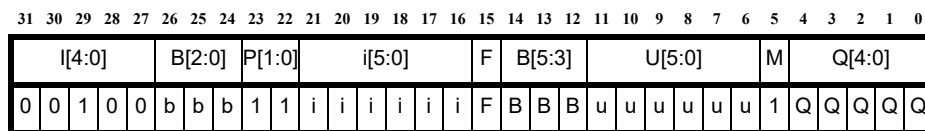
A value of 0x2F in the first operator field, *i*, indicates a SOP or ZOP format instruction, which is invalid for this operand mode and raises an [Illegal Instruction](#) exception.

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)
```

7.5.3.5 General Operations Conditional Register with Unsigned 6-bit Immediate

Figure 7-14 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)
```

7.5.4 Dual-operand Instructions, F32_GEN4

The DOP opcode assignments for F32_GEN4 are shown in Table 7-4. Subsequent sections contain similar tables for SOP and ZOP format instructions, and for other top-level formats (for example, F32_EXT5 and F32_EXT6). Each cell in these table represents a unique opcode slot that may be occupied by at most one instruction. Each table has eight columns, representing the eight value of the most significant three bits of the instruction opcode i[5:3]. Each table has eight pairs of rows, with each pair representing one of the eight values of the least significant three bits of the instruction opcode i[5:3]. The first row of each pair encodes instructions that have the <.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, raises an Illegal Instruction Exception.

A **Reserved** cell is one that is not allocated to an instruction, although attempting to execute the instruction with that opcode does not result in an exception. These cases indicate that the <.F> bit is ignored during the decoding of the instruction that is positioned directly above the **Reserved** cell.

Table 7-4 Opcode assignment for DOP instructions in major op-code 0x04 (F32_GEN4)

DOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ADD	MAX	BCLR	SUB2	J	LP	LD	SETEQ
	1	ADD.F	MAX.F	BCLR.F	SUB2.F	Reserved	Reserved		SETEQ.F
001	0	ADC	MIN	Illegal	SUB3	J.D	FLAG		SETNE
	1	ADC.F	MIN.F	BTST	SUB3.F	Reserved	KFLAG		SETNE.F
010	0	SUB	MOV	BXOR	MPYLO	JL	LR		SETLT
	1	SUB.F	MOV.F	BXOR.F	MPYLO.F	Reserved	Reserved		SETLT.F
011	0	SBC	Illegal	BMSK	MPYHI	JL.D	SR		SETGE
	1	SBC.F	TST	BMSK.F	MPYHI.F	Reserved	Reserved		SETGE.F
100	0	AND	DBNZ	ADD1	MPYHIU	BI	BMSKN		SETLO
	1	AND.F	CMP	ADD1.F	MPYHIU.F	Illegal	BMSKN.F		SETLO.F
101	0	OR	DBNZ.D	ADD2	MPYLOU	BIH	XBFU		SETHS
	1	OR.F	RCMP	ADD2.F	MPYLOU.F	Illegal	XBFU.F		SETHS.F
110	0	BIC	RSUB	ADD3	MPYW	LDI	Illegal		SETLE
	1	BIC.F	RSUB.F	ADD3.F	MPYW.F	Reserved	Illegal		SETLE.F
111	0	XOR	BSET	SUB1	MPYWU	AEX	SOP_FMT		SETGT
	1	XOR.F	BSET.F	SUB1.F	MPYWU.F	Reserved			SETGT.F

The F32_GEN4 format supports the majority of ALU operations in an orthogonal encoding of operand formats. It also contains certain jump and load instructions, on which are imposed a number of restrictions on the use of operand sub-formats. These **special formats** reserve certain operands or sub-formats that are not applicable to the instructions, or groups of instructions, encoded in those formats.

Table 7-5 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]				R				R				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]				1																																																									
																																	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																														
LP<cc>	0x28	0x4				R				0 1 1 0 1 1				i[5:0]				R				R				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																														
MOV<.cc><.F>	0x0A	0x4				B[2:0]				0 0 0 1 1 0 1 1				i[5:0]				F				B[5:3]				C[5:0]				Reserved																																	
BTST<.cc>	0x11 (F=1)																									U[5:0]								S[11:6]																													
TST<.cc>	0x0B (F=1)																									S[5:0]				S[11:6]																																	
																										C[5:0]				0				Q[4:0]																													
		U[5:0]				1																																																									
																																	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																														
FLAG<.cc>	0x29 (F=0)	0x4				R				0 0 0 1 1 0 1 1				i[5:0]				F				R				C[5:0]				Reserved																																	
KFLAG<.cc>	0x29 (F=1)																									U[5:0]								S[11:6]																													
																										S[5:0]				S[11:6]																																	
																										C[5:0]				0				Q[4:0]																													
		U[5:0]				1																																																									
																																	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																														
AEX<.cc>	0x27	0x4				B[2:0]				0 0 0 1 1 0 1 1				i[5:0]				R 1				B[5:3]				C[5:0]				Reserved																																	
CMP<.cc>	0x0C																									U[5:0]								S[11:6]																													
RCMP<.cc>	0x0D																									S[5:0]				S[11:6]																																	
																										C[5:0]				0				Q[4:0]																													
		U[5:0]				1																																																									
																																	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																														
LR	0x2A	0x4				B[2:0]				0 0 0 1 1 0				i[5:0]				R				B[5:3]				C[5:0]				Reserved																																	
SR	0x2B																									U[5:0]								S[11:6]																													
LDI	0x26																									S[5:0]				S[11:6]																																	
																																	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																														
BI, BIH	0x24, 0x25	0x4				R				0 0				i[5:0]				0				R				C[5:0]				Reserved																																	

7.5.5 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.

7.5.6 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.

7.5.7 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 6-11](#) on page 267, [Table 6-15](#) on page 269, and [Table 6-14](#) on page 268.

Figure 7-15 Load Register-Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			A	A	1	1	0	Z	Z	X	Di	B[5:3]			C[5:0]					A[5:0]							
0	0	1	0	0	b	b	b	a	a	1	1	0	Z	Z	X	D	B	B	B	C	C	C	C	C	C	A	A	A	A	A	A

The program counter (PCL) is not permitted to be the destination of a load instruction. Values, 0x3D and 0x3F, in the destination register field, A, raise an [Illegal Instruction](#) exception.

The loop count register (LP_COUNT) is not permitted to be the destination of a load instruction, A=0x3C, and raises an [Illegal Instruction](#) exception.

A value of 0x3 in the data size mode field, ZZ, raises an [Illegal Instruction](#) exception when LL64_OPTION is disabled.

The sign extension field, X, must not be set when the load is of 32-bit word data ZZ=0x0. This combination raises an [Illegal Instruction](#) exception.

Using incrementing addressing modes in combination with a long immediate values as the base register is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, raises an [Illegal Instruction](#) exception.

Syntax:

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

LD<zz><.x><.aa><.di> a,[b,limm]

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

7.5.8 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 7-6 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	

7.5.9 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 7-7 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	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	SLEEP	WLFC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	SWI	DSYNC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

Table 7-7 Opcode assignment for ZOP instructions in major op-code 0x04 (F32_GEN4)

011	0	SYNC	DMB	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	RTIE	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	BRK	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	SETI	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	CLRI	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

Syntax:

SLEEP

SLEEP u6

SLEEP c

SWI

SYNC

RTIE

BRK *(Encoded as REG_U6IMM, but the redundant REG_REG format is also valid. See [Table 7-3](#) on page 328)*

SETI c

SETI u6

CLRI c

CLRI u6

7.6 F32_EXT5

7.6.1 Dual-operand Instructions, F32_EXT5

Table 7-8 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5)

opcode		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111

Table 7-8 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5)

000	0	ASL multiple	REM	DMPYH	MPYD	VMPY2HWF	DMPYHWF	QMPYH	VADD4H
	1	ASL multiple.F	REM.F	DMPY.F	MPYD.F	SJLI	DMPYHWF.F	QMPYH.F	Illegal
001	0	LSR multiple	REMU	DMPYHU	MPYDU	VASL2H	VPACK2HL	QMPYHU	VSUB4H
	1	LSR multiple.F	REMU.F	DMPYHU.F	MPYDU.F	VASLS2H	VPACK2HM	QMPYHU.F	Illegal
010	0	ASR multiple	ASLS	DMACH	MACD	VASR2H	DMPYHF	DMPYWH	VADDSUB4H
	1	ASR multiple.F	ASLS.F	DMACH.F	MACD.F	VASRS2H	DMPYHF.F	DMPYWH.H	Illegal
011	0	ROR multiple	ASRS	DMACHU	MACDU	VLSR2H	DMPYHFR	DMPYWHU	VSUBADD4H
	1	ROR multiple.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	VSUBADDS2H	VMAC2HFR	SBCS.F		DMACWHU.F	Illegal

7.6.2 Single-operand Instructions, F32_EXT5

Table 7-9 Opcode assignment for SOP instructions in major op-code 0x05 (F32_EXT5)

SOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	SWAP	NORMH	ROL8	GETACC	VEXT2BHLF	VABS2H	SQRT	Illegal
		SWAP.F	NORMH.F	ROL8.F	Illegal	Illegal	Illegal	SQRT.F	Illegal
001	0	NORM	SWAPE	ROR8	NORMACC	VEXT2BHMF	VABSS2H	SQRTF	Illegal
	1	NORM.F	SWAPE.F	ROR8.F	Illegal	Illegal	Illegal	SQRTF.F	Illegal
010	0	SATH	LSL16	FFS	SATF	VREP2H	VNEG2H	Illegal	Illegal
	1	SATH.F	LSL16.F	FFS.F	SATF.F	Illegal	Illegal	Illegal	Illegal
011	0	RNDH	LSR16	FLS	Illegal	VREP2HM	VNEGS2H	Illegal	Illegal
	1	RNDH.F	LSR16.F	FLS.F	Illegal	Illegal	Illegal	Illegal	Illegal

Table 7-9 Opcode assignment for SOP instructions in major op-code 0x05 (F32_EXT5)

100	0	ABSSH	ASR16	Illegal	VPACK2HBL	VEXT2BHL	VNORM2H	Illegal	Illegal
	1	ABSSH.F	ASR16.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	ABSS	ASR8	Illegal	VPACK2HBM	VEXT2BHM	Illegal	Illegal	Illegal
	1	ABSS.F	ASR8.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	NEGHS	LSR8	Illegal	VPACK2HBLF	VSEXT2BHL	BSPEEK	Illegal	Illegal
	1	NEGHS.F	LSR8.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	NEGS	LSL8	Illegal	VPACK2HBMF	VSEXT2BHM	PSPOP	Illegal	ZOP_FMT
	1	NEGS.F	LSL8.F	Illegal	Illegal	Illegal	Illegal	Illegal	

7.6.3 Zero-operand Instructions, F32_EXT5

Table 7-10 Opcode assignment for ZOP instructions in major op-code 0x05 (F32_EXT5)

ZOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ASLACC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	ASLSACC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	FLAGACC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	MODIF	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

7.7 F32_EXT6

7.7.1 Dual-operand Instructions, F32_EXT6

Table 7-11 Opcode assignment for DOP instructions in major op-code 0x06 (F32_EXT6)

opcode		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	FSMUL	FCVT32	DIVF	DMACHBL	MPYWHFM	MACWHKL	FDMUL	FCVT64
	1	CMPLYHNFR	CMACCHNFR	DIVF.F	DMACHBL.F	MPYWHFM.F	MACWHKL.F	Illegal	Illegal
001	0	FSADD	FCVT32_64	VMAC2HNFR	DMACHBM	MPYWHFMR	MACWHKUL	FDADD	FCVT64_32
	1	CMPLYHFR	CMACCHFR	VMSUB2HNFR	DMACHBM.F	MPYWHFMR.F	MACWHKUL.F	Illegal	Illegal
010	0	FSSUB	MPYF	MPYDF	MSUBWHFLR	MACWHFM	MPYWHKL	FDSUB	Illegal
	1	CMPYCHNFR	MPYF.F	MPYDF.F	MSUBWHFLR.F	MACWHFM.F	MPYWHKL.F	Illegal	Illegal
011	0	VMSUB2HFR	MPYFR	MACDF	CMPLYHFR	MACWHFMR	MPYWHKUL	Illegal	Illegal
	1	FSCMP	MPYFR.F	MACDF.F	CBFLYHFOR	MACWHFMR.F	MPYWHKUL.F	FDCMP	Illegal
100	0	VMSUB2HF	MACF	MSUBWHFL	MPYWHL	MPYWHFL	MSUBWHFM	Illegal	Illegal
	1	FSCMPF	MACF.F	MSUBWHFL.F	MPYWHL.F	MPYWHFL.F	MSUBWHFM.F	Illegal	Illegal
101	0	FSMADD	MACFR	MSUBDF	MACWHL	MPYWHFLR	MSUBWHFMR	Illegal	Illegal
	1	CMPYCHFR	MACFR.F	MSUBDF.F	MACWHL.F	MPYWHFLR.F	MSUBWHFMR.F	FDCMPF	Illegal
110	0	FSMSUB	MSUBF	DMPYHBL	MPYWHUL	MACWHFL	Illegal	FDMADD	Illegal
	1	CMACHNFR	MSUBF.F	DMPYHBL.F	MPYWHUL.F	MACWHFL.F	Illegal	Illegal	Illegal
111	0	FSDIV	MSUBFR	DMPYHBM	MACWHUL	MACWHFLR	SOP_FMT	FDDIV	Illegal
	1	CMACHFR	MUSBFR.F	DMPYHBM.F	MACWHUL.F	MACWHFLR.F		Illegal	Illegal

7.7.2 Single-operand Instructions, F32_EXT6

Table 7-12 Opcode assignment for SOP instructions in major op-code 0x06 (F32_EXT6)

SOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	FSSQRT	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	FDSQRT	Illegal	Illegal	CBFLYHF1R	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

Table 7-12 Opcode assignment for SOP instructions in major op-code 0x06 (F32_EXT6)

100	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	ZOP_FMT
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	

7.7.3 Zero-operand Instructions, F32_EXT6

Table 7-13 Opcode assignment for ZOP instructions in major op-code 0x06 (F32_EXT6)

ZOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	SFLAG	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
		Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

7.8 APEX Extension Instruction Format [F32_APEX]

The F32_APEX top-level format, assigned to major opcode 0x07, is available for use by User Extension Instructions defined using the APEX extension mechanism.

Figure 7-16 using major opcode 0x07 as an example to show the syntax of op<.f> a,b,c encoding.

Figure 7-16 Extension ALU Operation, register-register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			P[1:0]		i[5:0]					F	B[5:3]			C[5:0]					A[5:0]								
0	0	1	1	1	b	b	b	0	0	i	i	i	i	i	i	F	B	B	B	C	C	C	C	C	C	A	A	A	A	A	A

7.8.1 Extension ALU Operation, Register with Unsigned 6-bit Immediate

Figure 7-17 shows the APEX encoding syntax of op<.f> a,b,u6 encoding.

Figure 7-17 Extension ALU Operation, register with unsigned 6-bit immediate

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			P[1:0]		i[5:0]					F	B[5:3]			U[5:0]					A[5:0]								
0	0	1	1	1	b	b	b	0	1	i	i	i	i	i	i	F	B	B	B	u	u	u	u	u	u	u	A	A	A	A	A

7.8.2 Extension ALU Operation, Register with Signed 12-bit Immediate

Using major opcode 0x05 as an example, Figure 7-18 shows the syntax of op<.f> b,b,s12 encoding.

Figure 7-18 Extension ALU Operation, register with signed 12-bit immediate

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			P[1:0]		i[5:0]					F	B[5:3]			S[5:0]					S[11:6]								
0	0	1	1	1	b	b	b	1	0	i	i	i	i	i	i	F	B	B	B	s	s	s	s	s	s	S	S	S	S	S	S

A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and raises an [Illegal Instruction](#) exception.

7.8.3 Extension ALU Operation, Conditional Register

Figure 7-19 shows the syntax of op<.cc><.f> b,b,c that is encoded.

Figure 7-19 Extension ALU Operation, conditional register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]				B[2:0]			P[1:0]		i[5:0]					F	B[5:3]			C[5:0]					M	Q[4:0]							
0	0	1	1	1	b	b	b	1	1	i	i	i	i	i	i	F	B	B	B	C	C	C	C	C	C	0	Q	Q	Q	Q	Q

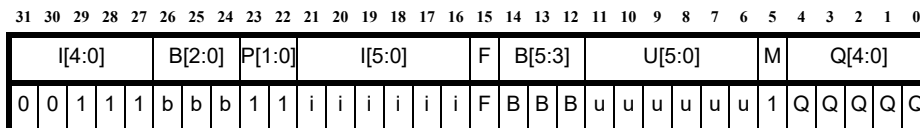
A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and raises an [Illegal Instruction](#) exception.

Values 0x10 to 0x1F in the condition code field, Q, raises an **Illegal Instruction** exception unless the Q field specifies a condition code defined as an extension condition.

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

Using major opcode 0x05 as an example, [Figure 7-20](#) shows the syntax of op<.cc><.f> b,b,u6 encoding.

Figure 7-20 Extension ALU Operation, cc register with unsigned 6-bit immediate



A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and raises an **Illegal Instruction** exception.

Values 0x10 to 0x1F in the condition code field, Q, raise an **Illegal Instruction** exception unless the Q field specifies a condition code defined as an extension condition.

7.8.5 FastMath Extension Pack Instructions

[Table 7-14](#) lists the FastMath Pack Extension Instructions. If this extension is available with your processor, the instructions are available in the *Part: FastMath Pack Extension* in the *ARCV2 ISA Programmer's Reference Manual*.

Table 7-14 List of FastMath extension pack instructions

Instruction	Operation
FMP_ADDS	32-bit signed addition with result saturation
FMP_RNDH	Round and saturate a 32-bit signed 2's complement value to a 16-bit value
FMP_SATH	Saturate a 32-bit signed 2's complement integer to a 16-bit value
FMP_DIVF	Fractional division of Q31 numerator X, by Q31 divisor Y, where $\text{abs}(X) < \text{abs}(Y)$ or $X = -Y$, and $Y \neq 0$.
FMP_DIVF15	Fractional division of Q15 numerator X, by Q15 divisor Y, where $\text{abs}(X) < \text{abs}(Y)$ or $X = -Y$, and $Y \neq 0$.
FMP_RECIP	Computes the reciprocal of a Q31 fraction X, provided $X \neq 1$.
FMP_RECIP15	Computes the reciprocal of a Q15 fraction X, provided $X \neq 1$.
FMP_SQRTF	Computes the square root of the Q31 fractional argument X, when $X \geq 0$.
FMP_SQRTF15	Computes the square root of the Q15 fractional argument X, when $X \geq 0$.
FMP_COS	Computes the cosine function on the Q31 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting cosine value is returned as the Q31 result.

Table 7-14 List of FastMath extension pack instructions

Instruction	Operation
FMP_COS15	Computes the cosine function on the Q15 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting cosine value is returned as the Q15 result.
FMP_SIN	Computes the sine function on the Q31 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting sine value is returned as the Q31 result.
FMP_SIN15	Computes the sine function on the Q15 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting sine value is returned as the Q15 result.
FMP_ATAN	Computes the inverse tangent function on the Q31 fractional input operand. The input operand represents a real-valued tangent in the range $[-1,0)$ and the Q31 result represents the angle expressed in radians divided by Pi, for which the tangent is equal to the input operand.
FMP_ATAN15	Computes the inverse tangent function on the Q15 fractional input operands. The input operand represents a real-valued tangent in the range $[-1,0)$ and the Q15 result represents the angle expressed in radians divided by Pi, for which the tangent is equal to the input operand.
FMP_LOG2	Computes the base-2 logarithm of the Q31 fractional input operand X, when X is in the range $[0.5,1)$. The result $Y = \log_2(X)$ is a Q31 fraction in the range $[-1,0)$.
FMP_LOG215	Computes the base-2 logarithm of the Q15 fractional input operand X, when X is in the range $[0.5,1)$. The result $Y = \log_2(X)$ is a Q15 fraction in the range $[-1,0)$.
FMP_EXP2	Computes the base-2 exponentiation of the Q31 fractional input operand X, when X is in the range $[-1,0)$. The result $Y = 2^X$ is a Q31 fraction in the range $[0.5,1)$.
FMP_EXP215	Computes the base-2 exponentiation of the Q15 fractional input operand X, when X is in the range $[-1,0)$. The result $Y = 2^X$ is a Q15 fraction in the range $[0.5,1)$.

8

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 6-4](#) on page [261](#) 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 6-17](#) on page [270](#), and section “[Encoding Notation](#)” on page [262](#) describes the encoding notation used when describing instruction encodings in this chapter.

This chapter describes each of the following 16-bit top-level formats:

- [Compact Move/Load \[F16_COMPACT\]](#)
- [Compact Load/Add/Sub \[F16_LD_ADD_SUB\]](#)
- [Compact Load/Store \[F16_LD_ST_1\]](#)
- [Indexed Jump or Execute \[F16_JLI_EI\]](#)
- [Load / Add Register-Register \[F16_LD_ADD_RR\]](#)
- [Add/Sub/Shift Register-Immediate \[F16_ADD_IMM\]](#)
- [Dual Register Operations \[F16_OP_HREG\]](#)
- [General Register Format Instructions \[F16_GEN_OP\]](#)
- [16-bit Load and Store Formats with Offset](#)
- [Shift/Subtract/Bit Immediate \[F16_SH_SUB_BIT\]](#)
- [Stack-based Operations \[F16_SP_OPS\]](#)
- [Load/Add GP-Relative \[F16_GP_LD_ADD\]](#)
- [Load PCL-Relative \[F16_PCL_LD\]](#)
- [Move Immediate \[F16_MV_IMM\]](#)
- [ADD/CMP Immediate \[F16_OP_IMM\]](#)
- [Branch on Compare Register with Zero \[F16_BCC_REG\]](#)
- [Branch Conditionally \[F16_BCC\]](#)
- [Branch and Link Unconditionally \[F16_BL\]](#)

Each section provides details of the sub-formats, and describes how instructions and their operands are encoded in those sub-formats.

For a summary of the instructions discussed in this chapter, and an explanation of the functions they perform, see [Chapter 6, “Instruction Set Summary”](#).

For full details on the available encodings and syntax for each instruction, see the relevant page for each instructions in [Chapter 9, “Instruction Set Details”](#).

8.1 Compact Move/Load [F16_COMPACT]

Extension Group: BASELINE

Figure 8-1 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 8-2 Compact Load Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					U	r[1:0]		h[2:0]			U[3:2]		i	H[4:3]	
0	1	0	0	0	U[4]	r	r	h	h	h	u	u	1	H	H

The compact load encoded in this format allows a load from h-register base address, plus a u5 scaled offset, into one of R0, R1, R2 or R3.

Syntax:

LD_S R0/1/2/3, [h,u5]

8.2 Compact Load/Add/Sub [F16_LD_ADD_SUB]

Extension Group: CODE_DENSITY

Figure 8-3 Compact Load and Subtract Format

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

The first sub-format of opcode 0x09 encodes additional LD_S.AS and SUB_S instructions in a compact format to improve code density.

Syntax:

LD_S.AS a, [b, c]

SUB_S a, b, c

Figure 8-4 Compact Add Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
l[4:0]					b[2:0]			r	U[5:3]			1	u[2:0]		
0	1	0	0	1	b	b	b	r	U	U	U	1	u	u	u

The second sub-format of opcode 0x09 encodes additional ADD_S instructions, with R0 or R1 as their destination register, in a compact form to improve code density.

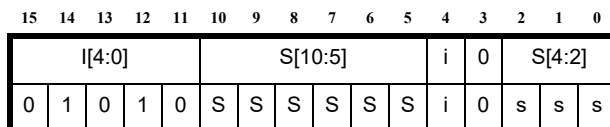
Syntax:

ADD_S r, b, u6 r can be R0 or R1

8.3 Compact Load/Store [F16_LD_ST_1]

Extension Group: CODE_DENSITY

Figure 8-5 Compact Load and Store Format



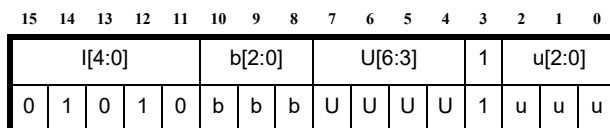
The first sub-format in opcode 0x0A encodes two additional GP-relative load/store instructions for use when accessing the most frequently occurring small data objects.

Syntax:

LD_S R1, [GP, s11]

ST_S R0, [GP, s11]

Figure 8-6 Compact Load Indexed Format



The second sub-format in opcode 0x0A encodes the Load Indexed instruction (LDI_S). This instruction loads from a table of 128 32-bit word entries, given the table index in the instruction. The intent of this instruction is to create a global constant pool to store the most commonly-used L IMM constants of the program. In this way, such constants only need to appear once in the linked executable, and such a constant can always be loaded into a compressed register using a 16-bit instruction.

LDI_BASE is an auxiliary register containing the address of the data table, which must be aligned on a 4-byte boundary.

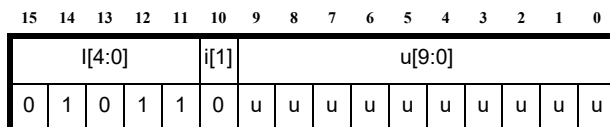
Syntax:

LDI_S b, [u7]

8.4 Indexed Jump or Execute [F16_JLI_EI]

Extension Group: CODE_DENSITY

Figure 8-7 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:

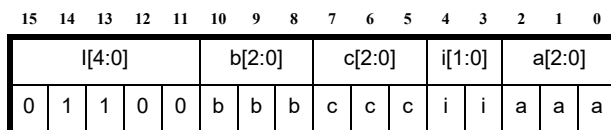
EI_S u10

Table 8-1 16-Bit Indexed Jump and Link Operations

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

8.5 Load /Add Register-Register [F16_LD_ADD_RR]

Figure 8-8 Load and Add Register-Register Format



Syntax:

LD_S a, [b, c]

LDB_S a, [b, c]

LDH_S a, [b, c]

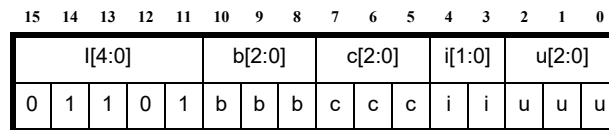
ADD_S a, b, c

Table 8-2 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

8.6 Add/Sub/Shift Register-Immediate [F16_ADD_IMM]

Figure 8-9 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 8-3 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.

8.7 Dual Register Operations [F16_OP_HREG]

Figure 8-10 Dual Register with a 3-bit b Register Format

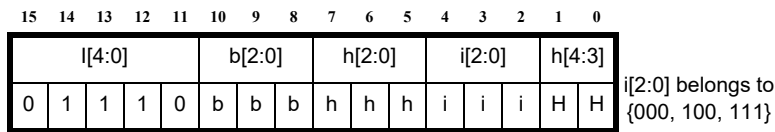
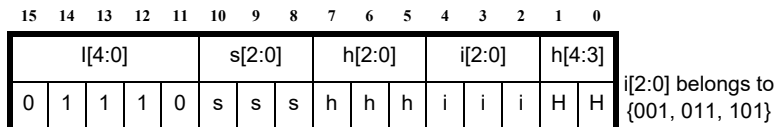


Figure 8-11 Dual Register with a 3-bit Biased Signed Integer Format



Format 0x0E encodes eight distinct operators, each with two register operands. One of the two operands is a 3-bit 'b' register, or a 3-bit biased signed integer 's3', and the other operand is a 5-bit 'h' register specifier. The 'b' register selects one of the registers normally accessible with a 3-bit register field within a 16-bit instruction encoding. The 'h' register is capable of selecting any register in the range r0 – r28, and also r31. The 'h' field encoding for r30 indicates the use of long immediate data. An 'h' field encoding of r29 is not allowed.

The s3 field encodes the signed integers values -1, 0, 1, 2, 3, 4, 5, 6 using binary codes 111, 000, 001, 010, 011, 100, 101, and 110, respectively.

Syntax:

ADD_S	b, b, h	(h != 0x1e)
ADD_S	b, b, limm	(h = 0x1e)
ADD_S	h, h, s3	(h != 0x1e)
ADD_S	0, limm, s3	(h = 0x1e)
MOV_S	h, s3	(h != 0x1e)
MOV_S	0, s3	(h = 0x1e)
MOV_S.NE	b, h	(h != 0x1e)
MOV_S.NE	b, limm	(h = 0x1e)
CMP_S	b, h	(h != 0x1e)
CMP_S	b, limm	(h = 0x1e)
CMP_S	h, s3	(h != 0x1e)
CMP_S	limm, s3	(h = 0x1e)

Table 8-4 16-Bit MOV/CMP/ADD with High Register

Sub-opcode i field (3 bits)	Instruction	Operation	Description
0x00	ADD_S	$b \leftarrow b + h$	Add
0x01	ADD_S	$h \leftarrow h + s3$	Add
0x02		Illegal Instruction Error	Reserved
0x03	MOV_S	$h \leftarrow s3$	Move
0x04	CMP_S	$b - h$	Compare
0x05	CMP_S	$h - s3$	Compare
0x06		Illegal Instruction Error	Reserved
0x07	MOV_S.NE	if (!Z) $b \leftarrow h$	Move if STATUS32[Z] == 0

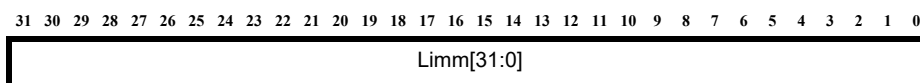
8.7.1 Long Immediate Operands in Format 0x0E

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 8-12 Long Immediate Operand Format

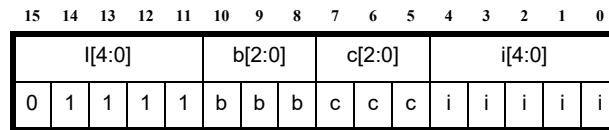
Syntax:

limm

8.8 General Register Format Instructions [F16_GEN_OP]

8.8.1 DOP-format General Operations

Figure 8-13 Compact General Operations Register-Register Format



Syntax:

op_S b,b,c

op_S b,c

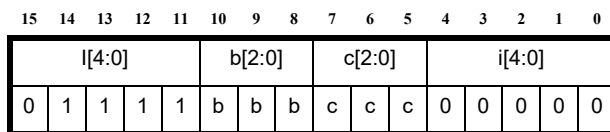
Table 8-5 DOP_format 16-Bit General Operations

Sub-opcode i field (5 bits)	Instruction	Operation	Description
0x00	SOPs	c field is sub-opcode2	See Table 8-6 on page 355.
0x01		Illegal Instruction	Reserved
0x02	SUB_S	$b \leftarrow b - c$	Subtract
0x03		Illegal Instruction	Reserved
0x04	AND_S	$b \leftarrow b \text{ and } c$	Logical bitwise AND
0x05	OR_S	$b \leftarrow b \text{ or } c$	Logical bitwise OR
0x06	BIC_S	$b \leftarrow b \text{ and not } c$	Logical bitwise AND with invert
0x07	XOR_S	$b \leftarrow b \text{ exclusive-or } c$	Logical bitwise exclusive-OR
0x08		Illegal Instruction	Reserved
0x09	MPYW_S	$b \leftarrow b * c$	Multiply 16-bit half-words
0x0A	MPYUW_S	$b \leftarrow b * c$ (unsigned)	Multiply unsigned 16-bit half-words
0x0B	TST_S	$b \text{ and } c$	Test
0x0C	MPY_S	$b \leftarrow b * c$	32 X 32 Multiply (least-significant half)
0x0D	SEXB_S	$b \leftarrow \text{sexb } c$	Sign extend byte
0x0E	SEXH_S	$b \leftarrow \text{sexw } c$	Sign extend 16-bit half-word
0x0F	EXTB_S	$b \leftarrow \text{extb } c$	Zero extend byte

Table 8-5 DOP_format 16-Bit General Operations (Continued)

Sub- opcode i field (5 bits)	Instruction	Operation	Description
0x10	EXTH_S	$b \leftarrow \text{extw } c$	Zero extend 16-bit half-word
0x11	ABS_S	$b \leftarrow \text{abs } c$	Absolute
0x12	NOT_S	$b \leftarrow \text{not } c$	Logical NOT
0x13	NEG_S	$b \leftarrow \text{neg } c$	Negate
0x14	ADD1_S	$b \leftarrow b + (c \ll 1)$	Add with left shift by 1
0x15	ADD2_S	$b \leftarrow b + (c \ll 2)$	Add with left shift by 2
0x16	ADD3_S	$b \leftarrow b + (c \ll 3)$	Add with left shift by 3
0x17		Illegal Instruction	Reserved
0x18	ASL_S	$b \leftarrow b \text{ asl } c$	Multiple arithmetic shift left
0x19	LSR_S	$b \leftarrow b \text{ lsr } c$	Multiple logical shift right
0x1A	ASR_S	$b \leftarrow b \text{ asr } c$	Multiple arithmetic shift right
0x1B	ASL_S	$b \leftarrow c + c$	Arithmetic shift left by one
0x1C	ASR_S	$b \leftarrow c \text{ asr } 1$	Arithmetic shift right by one
0x1D	LSR_S	$b \leftarrow c \text{ lsr } 1$	Logical shift right by one
0x1E	TRAP_S	Trap	Raise Exception
0x1F	BRK_S	Break	Break (Encoding is 0x7FFF)

8.8.2 SOP-format 16-bit General Operations

Figure 8-14 Compact Single Operand, Jumps, Special Formats

Syntax:

op_S b

op_S b,b

J_S<.d> [b]

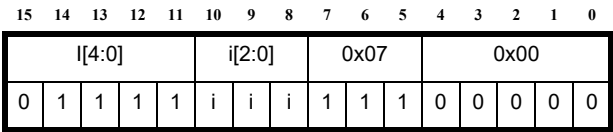
JL_S<.d> [b]
SUB_S.ne b,b,b

Table 8-6 16-Bit Single Operand Instructions

Sub- opcode2 c field (3 bits)	Instruction	Operation	Description
0x00	J_S	pc ← b	Jump
0x01	J_S.D	pc ← b	Jump delayed
0x02	JL_S	blink ← next_pc; pc ← b	Jump and link
0x03	JL_S.D	blink ← next_pc; pc ← b	Jump and link delayed
0x04		Illegal Instruction	Reserved
0x05		Illegal Instruction	Reserved
0x06	SUB_S.NE	if (flags.Z==0) then b ← b – b	If Z flag is 0, clear register
0x07	ZOP s	b field is sub-opcode3	See Table 8-7 on page 356.

8.8.3 ZOP-format 16-bit General Operations

Figure 8-15 Compact Zero Operand Instructions



Syntax:

op_S

NOP_S

UNIMP_S

J_S<.d> [blink]

JEQ_S [blink]

JNE_S [blink]

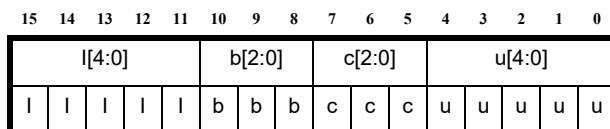
Table 8-7 16-Bit Zero Operand Instructions

Sub- opcode3 b field (3 bits)	Instruction	Operation	Description
0x00	NOP_S	nop	No operation
0x01	UNIMP_S	Illegal Instruction	Unimplemented Instruction
0x02	SWI_S	Software Interrupt	Raise a software interrupt exception.
0x03		Illegal Instruction	Reserved
0x04	JEQ_S [blink]	pc ← blink	Jump using blink register if equal
0x05	JNE_S [blink]	pc ← blink	Jump using blink register if not equal
0x06	J_S [blink]	pc ← blink	Jump using blink register
0x07	J_S.D [blink]	pc ← blink	Jump using blink register delayed

8.9 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 8-16 Compact Load/Store with Offset Format



Syntax:

LD_S c, [b, u7] (*u7 must be 32-bit aligned*)

LDB_S c, [b, u5]

LDH_S c, [b, u6] (*u6 must be 16-bit aligned*)

LDH_S.X c, [b, u6] (*u6 must be 16-bit aligned*)

ST_S c, [b, u7] (*u7 must be 32-bit aligned*)

STB_S c, [b, u5]

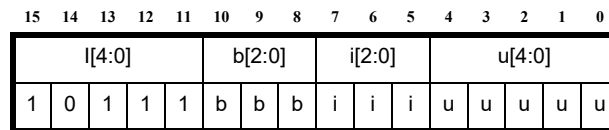
STH_S c, [b, u6] (*u6 must be 16-bit aligned*)

Table 8-8 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

8.10 Shift/Subtract/Bit Immediate [F16_SH_SUB_BIT]

Figure 8-17 Compact Shift/Sub Bit Immediate Format



Syntax:

SUB_S b, b, u5

BSET_S b, b, u5

BCLR_S b, b, u5

BMSK_S b, b, u5

BTST_S b, u5

ASL_S b, b, u5

LSR_S b, b, u5

ASR_S b, b, u5

Table 8-9 16-Bit Shift/SUB/Bit Immediate

Sub-opcode2 i field (3 bits)	Instruction	Operation	Description
0x00	ASL_S	$b \leftarrow b \text{ asl } u5$	Multiple arithmetic shift left
0x01	LSR_S	$b \leftarrow b \text{ lsr } u5$	Multiple logical shift right
0x02	ASR_S	$b \leftarrow b \text{ asr } u5$	Multiple arithmetic shift right
0x03	SUB_S	$b \leftarrow b - u5$	Subtract
0x04	BSET_S	$b \leftarrow b \text{ or } 1 \ll u5$	Bit set
0x05	BCLR_S	$b \leftarrow b \text{ and not } 1 \ll u5$	Bit clear
0x06	BMSK_S	$b \leftarrow b \text{ and } ((1 \ll (u5+1)) - 1)$	Bit mask
0x07	BTST_S	$b \text{ and } 1 \ll u5$	Bit test

8.11 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 8-10. 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 8-10 Opcodes for Stack-based Operations

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Encoded Instructions		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	u
LDB_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	0	1	u	u	u	u	u
ST_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	1	0	u	u	u	u	u
STB_S	b, [SP, u7]	1	1	0	0	0	b	b	b	0	1	1	u	u	u	u	u
ADD_S	b, SP, u7	1	1	0	0	0	b	b	b	1	0	0	u	u	u	u	u
ADD_S	SP, SP, u7	1	1	0	0	0	0	0	0	1	0	1	u	u	u	u	u
SUB_S	SP, SP, u7	1	1	0	0	0	0	0	1	1	0	1	u	u	u	u	u
POP_S	b	1	1	0	0	0	b	b	b	1	1	0	0	0	0	0	1
POP_S	BLINK	1	1	0	0	0	R	R	R	1	1	0	1	0	0	0	1
PUSH_S	b	1	1	0	0	0	b	b	b	1	1	1	0	0	0	0	1
PUSH_S	BLINK	1	1	0	0	0	R	R	R	1	1	1	1	0	0	0	1
LEAVE_S	u7	1	1	0	0	0	U	U	U	1	1	0	u	u	u	u	0
ENTER_S	u6	1	1	0	0	0	0	U	U	1	1	1	u	u	u	u	0

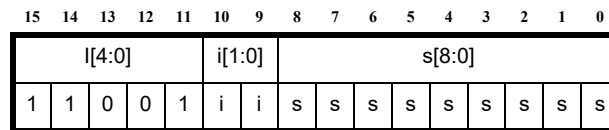
The ENTER_S and LEAVE_S instructions are available only when the code_density extension pack is configured in the architecture. If an ENTER_S or LEAVE_S instruction is executed on a build without the code_density extension pack, an Illegal Instruction Error exception is raised.

This format contains a number of unused encodings. There are, for example, 6 unused encodings in sub-opcode 5, for values of $b[2:0]$ in the range 2 to 7. Any attempt to execute an unused encoding raises an Illegal Instruction Error exception.

Following the normal rule for the use of reserved fields, any non-zero value in a reserved field is ignored. However, reserved bits must be set to 0 when assembling instructions.

8.12 Load/Add GP-Relative [F16_GP_LD_ADD]

Figure 8-18 Compact Load/Add GP-Relative Format



The offset (s[8:0]) is shifted accordingly to provide the appropriate data size alignment.

Syntax:

LD_S r0, [GP, s11] *(32-bit aligned offset)*

LDB_S r0, [GP, s9] *(8-bit aligned offset)*

LDH_S r0, [GP, s10] *(16-bit aligned offset)*

ADD_S r0, GP, s11 *(32-bit aligned offset)*

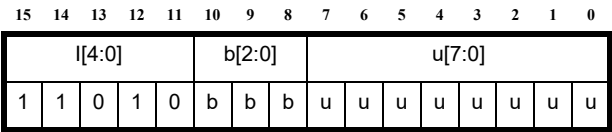
Table 8-11 16-Bit GP Relative Instructions

Sub- opcode i field (2 bits)	Instruction	Operation	Description
0x00	LD_S	$r0 \leftarrow \text{mem}[GP + s11].l$	Load GP-relative (32-bit aligned) to r0
0x01	LDB_S	$r0 \leftarrow \text{mem}[GP + s9].b$	Load unsigned byte GP-relative (8-bit aligned) to r0
0x02	LDH_S	$r0 \leftarrow \text{mem}[GP + s10].w$	Load unsigned 16-bit half-word GP-relative (16-bit aligned) to r0
0x03	ADD_S	$r0 \leftarrow GP + s11$	Add GP-relative (32-bit aligned) to r0

8.13 Load PCL-Relative [F16_PCL_LD]

The offset (u[7:0]) is shifted accordingly to provide the appropriate 32-bit data size alignment.

Figure 8-19 Compact Load PCL Relative Format

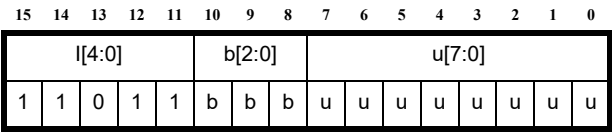


Syntax:

```
LD_S b, [PCL, u10] (32-bit aligned offset)
```

8.14 Move Immediate [F16_MV_IMM]

Figure 8-20 Compact Move Immediate Format

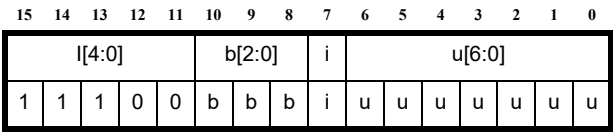


Syntax:

```
MOV_S b, u8
```

8.15 ADD/CMP Immediate [F16_OP_IMM]

Figure 8-21 Compact ADD/CMP Immediate Format



Syntax:

ADD_S b, b, u7

CMP_S b, u7

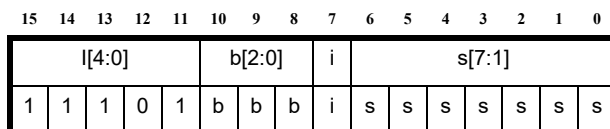
Table 8-12 16-Bit ADD/CMP Immediate

Sub- opcode			
i field (1 bit)	Instruction	Operation	Description
0x00	ADD_S	$b \leftarrow b + u7$	Add
0x01	CMP_S	$b - u7$	Compare

8.16 Branch on Compare Register with Zero [F16_BCC_REG]

The target address is 16-bit aligned.

Figure 8-22 Compact Branch on Compare Register with Zero Format



Syntax:

BREQ_S b, 0, s8

BRNE_S b, 0, s8

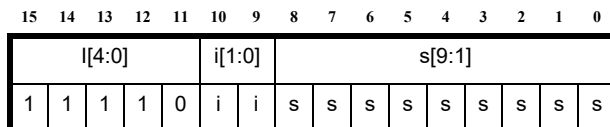
Table 8-13 16-Bit Branch on Compare

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

8.17 Branch Conditionally [F16_BCC]

The target address is 16-bit aligned.

Figure 8-23 Compact Branch Conditionally Format



Syntax:

B_S s10

BEQ_S s10

BNE_S s10

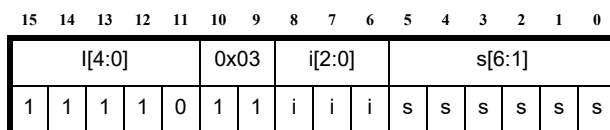
Table 8-14 16-Bit Branch, Branch Conditionally

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	B_S	PC = (PCL+s10)	Branch always
0x01	BEQ_S	if (Z) PC = (PCL+s10)	Branch if equal
0x02	BNE_S	if (/Z) PC = (PCL+s10)	Branch if not equal
0x03	Bcc_S	if (cc) PC = (PCL+s10)	See Bcc_S

8.17.1 Branch Conditionally with cc Field, 0x1E, [0x03, 0x00 – 0x07]

The target address is 16-bit aligned.

Figure 8-24 Branch Conditionally with cc Field Format



Syntax:

BGT_S s7

BGE_S s7

BLT_S s7

BLE_S s7

BHI_S s7

BHS_S s7

BLO_S s7

BLS_S s7

Table 8-15 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 6-7](#).

8.18 Branch and Link Unconditionally [F16_BL]

The target address can only target 32-bit aligned instructions.

Figure 8-25 Compact Branch and Link Unconditionally Format



Syntax:

BL_S s13

9

Instruction Set Details

This chapter lists the available instruction set in alphabetic order. The syntax and encoding examples list full syntax for each instruction, but excludes the redundant encoding formats. A full list of encoding formats can be found in [“Instruction Set Summary”](#).

Both 32-bit and 16-bit instruction encodings are available in the ARCV2 ISA and are indicated using particular suffixes on the instruction as illustrated by the following syntax:

OP implies 32-bit encoding

OP_L indicates 32-bit encoding.

OP_S indicates 16-bit encoding

If no suffix is used on the instruction then the implied instruction format is 32 bits.

[Table 6-17](#) on page 270 lists the syntax conventions. [Table 6-5](#) on page 262 and [Table 6-6](#) on page 263 lists the encoding notation.

[Table 9-1](#) summarizes the 32-bit format instructions alongside the 16-bit format instructions supported by the ARCV2 ISA.

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [Dual-operand Instructions, F32_GEN4](#).

9.1 Instruction List

Table 9-1 List of Instructions

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
ABS	Absolute value	ABS_S	Absolute value
ADC	Add with carry		
ADD	Add	ADD_S	Add
ADD1	Add with left shift by 1 bit	ADD1_S	Add with left shift by 1 bits
ADD2	Add with left shift by 2 bits	ADD2_S (see ADD2)	Add with left shift by 2 bits
ADD3	Add with left shift by 3 bits	ADD3_S (see ADD3)	Add with left shift by 3 bits
AEX	Swap contents of auxiliary register with a 32-bit core register		
AND	Logical AND	AND_S (see AND)	Logical AND
ASL	Arithmetic Shift Left	ASL_S (see ASL)	Arithmetic Shift Left
ASR	Arithmetic Shift Right	ASR_S (see ASR)	Arithmetic Shift Right
ASR16	Arithmetic Shift Right by 16		
ASR8	Arithmetic Shift Right by 8		
B	Branch unconditionally	B_S	Branch unconditionally
BBIT0	Branch if bit equal to 0		
BBIT1	Branch if bit equal to 1		
Bcc	Branch if condition true	Bcc_S	Branch if condition true
BCLR	Clear specified bit (to 0)	BCLR_S (see BCLR)	Clear specified bit (to 0)
BI	Branch Indexed, 32-bit full-word table		
BIH	Branch Indexed, 16-bit Half-word table		
BIC	Bit-wise inverted AND	BIC_S (see BIC)	Bit-wise inverted AND
BLcc	Branch and Link	BL_S (see BLcc)	Branch and Link

Table 9-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
BMSK	Bit Mask	BMSK_S (see BMSK)	Bit Mask
BMSKN	Bit Mask Negated		
BRcc	Branch on compare	BRcc_S (see BRcc)	Branch on compare
BRK	Break (halt) processor	BRK_S (see BRK)	Break (halt) processor
BSET	Set specified bit (to 1)	BSET_S (see BSET)	Set specified bit (to 1)
BSPEEK	Peek the head of the bitstream		
BSPOP	Pop head of the bitstream		
BSPUSH	Push to the tail of the bitstream		
BTST	Test value of specified bit	BTST_S (see BTST)	Test value of specified bit
BXOR	Bit XOR		
CLRI	Clear Interrupt Enable		
CMP	Compare	CMP_S (see CMP)	Compare
DBNZ	Signed integer Divide		
DIV	Signed integer Divide		
DIVU	Unsigned integer Divide		
DMACH	Dual 16x16 multiply and accumulate		
DMACHU	Dual unsigned 16x16 multiply and accumulate		
DMACWH	Dual 32x16 multiply and accumulate		
DMACWHU	Dual unsigned 32x16 multiply and accumulate		
DMB	Data memory barrier instruction		
DMPYH	Dual 16x16 multiplication		
DMPYHU	Dual unsigned 16x16 multiplication		
DMPYWH	Dual 32x16 multiplication		

Table 9-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
DMPYWHU	Dual unsigned 32x16 multiplication		
DSYNC	Synchronize instruction		
		EL_S	Execute Indexed
		ENTER_S	Function prolog Sequence
EX	Atomic Exchange		
EXTB	Zero-extend byte to word	EXTB_S (see EXTB)	Zero-extend byte
EXTH	Zero-extend 16-bit half-word to word	EXTH_S (see EXTH)	Zero-extend 16-bit half-word
FCVT32	Conversion between 32-bit data formats		
FCVT32_64	Conversion from a 32-bit data format to a 64-bit data format		
FCVT64	Conversion between 64-bit data formats		
FCVT64_32	Conversion from a 64-bit to a 32-bit data format		
FDABS	Double-precision floating-point absolute operation		
FDADD	Double-precision floating-point addition		
FDCMP	Double-precision floating-point comparison		
FDCMPF	Double-precision floating-point comparison		
FDDIV	Double-precision floating-point division		
FDMADD	Double-precision floating-point fusedMultiplyAdd A fusedMultiplyAdd is a floating-point multiply-add operation performed in one step, with a single rounding.		

Table 9-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
FDMSUB	Double-precision floating-point fusedMultiplySubtract A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding.		
FDMUL	Double-precision floating-point multiplication		
FDNEG	Double-precision floating-point negation		
FDSQRT	Double-precision floating-point square-root operation		
FDSUB	Double-precision floating-point subtraction		
FFS	Find First Set		
FLAG	Write to Status Register		
FLS	Find Last Set		
FSABS	Single-precision floating-point absolute operation		
FSADD	Single-precision floating-point addition		
FSCMP	Single-precision floating-point comparison		
FSCMPF	Single-precision floating-point comparison		
FSDIV	Single-precision floating-point division		
FSMADD	Single-precision floating-point fusedMultiplyAdd A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding.		

Table 9-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
FSMSUB	Single-precision floating-point fusedMultiplySubtract A fusedMultiplySubtract is a floating-point multiply-subtract operation performed in one step, with a single rounding.		
FSMUL	Single-precision floating-point multiplication		
FSNEG	Single-precision floating-point negation		
FSSQRT	Single-precision floating-point square-root		
FSSUB	Single-precision floating-point subtraction		
Jcc	Jump	J_S (see Jcc)	Jump
JL	Jump	JL_S (see Jcc)	Jump and Link
		JLI_S	Jump and Link Indexed
KFLAG	Write to Status Register in kernel mode		
LD LDH LDW LDB LDD	Load from memory	LD_S (see LD LDH LDW LDB LDD)	Load from memory
LDI (see LDI , LDI_S)	Load Indexed	LDI_S (see LDI , LDI_S)	Load Indexed
		LEAVE_S	Function Epilog Sequence
LLOCK	Load locked		
LLOCKD	Load locked on 64-bit data (ARC-32 ABI)		
LP<cc>	Loop (zero-overhead loops)		
LR	Load from Auxiliary memory		
LSL16	Logical Shift Left 16		
LSL8	Logical Shift Left 8		
LSR	Logical Shift Right	LSR_S (see LSR)	Logical Shift Right
LSR16	Logical Shift Right 16		

Table 9-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
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 MOV)	Move (copy) to register
MPY, MPY_S	32 x 32 Signed Multiply (lsw)	MPY_S (see MPY, MPY_S)	32 x 32 Signed Multiply (lsw)
MPYM MPYH	32 x 32 Signed Multiply (msw)		
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 MPYUW, MPYUW_S)	
MPYW, MPYW_S	16 X 16 signed multiply	MPYW_S (see MPYW, MPYW_S)	16 x16 signed multiply
NEG	Negate	NEG_S (see NEG)	Negate
NOP	No operation	NOP_S (see NOP)	No operation
NORM	Normalize to 32 bits		
NORMH, NORMW	Normalize to 16 bits		
NOT	Logical bit inversion	NOT_S (see NOT)	Logical bit inversion
OR	Logical OR	OR_S (see OR)	Logical OR
		POP_S	Restore register from stack

Table 9-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
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
QMACH	Quad 16x16 multiply and accumulate		
QMACHU	Quad unsigned 16x16 multiply and accumulate		
QMPYH	Quad 16x16 multiplication		
QMPYHU	Quad unsigned 16x16 multiplication		
RCMP	Reverse Compare		
REM	Signed Integer Remainder		
REMU	Unsigned Integer Remainder		
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		

Table 9-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
SEXB	Sign-extend byte to word	SEXB_S (see SEXB)	Sign-extend byte
SEXH, SEXW	Sign-extend half-word to word	SEXH_S (see SEXH, SEXW)	Sign-extend 16-bit half-word
SLEEP	Put processor in sleep state		
SR	Store to Auxiliary memory		
ST ST STH STB STD	Store to memory	ST_S (see ST ST STH STB STD)	Store to memory
SUB	Subtract	SUB_S (see SUB)	Subtract
SUB1	Subtract with left shift by 1 bit		
SUB2	Subtract with left shift by 2 bits		
SUB3	Subtract with left shift by 3 bits		
SWAP	Swap 16-bit register halves		
SWAPE	Swap byte ordering		
SWI	Software interrupt	SWI_S (see SWI)	Software interrupt
SYNC	Synchronize		
		TRAP_S	Trap to system call
TST	Test	TST_S (see TST)	Test
VADD2	Dual 32-bit vector addition		
VADD2H	Dual 16-bit vector addition		
VADD4H	Quad 16-bit vector addition		
VADDSUB	Dual 32-bit vector add and subtract		
VADDSUB2H	Dual 16-bit vector add and subtract		
VADDSUB4H	Quad 16-bit vector add and subtract		
VMAC2H	Dual 16x16 vector multiplication and accumulation		
VMPY2H	Dual 16x16 vector multiplication		

Table 9-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	Operation	Instruction	Operation
VMPY2HU	Dual unsigned 16x16 vector multiplication		
VSUB2	Dual 32-bit vector subtraction		
VSUB2H	Two way vector subtract 16 bits		
VSUB4H	Quad 16-bit vector subtraction		
VSUBADD	Dual 32-bit vector subtract and add		
VSUBADD2H	Dual 16-bit vector subtract and add		
VSUBADD4H	Quad 16-bit vector subtract and add		
WEVT	Enter sleep state and wait on event		
WLFC	Enter Sleep State to reduce dynamic power during busy-waiting loops		
XBFU	Extract unsigned bit-field		
XOR	Logical Exclusive-OR	XOR_S (see XOR)	Logical Exclusive-OR
		UNIMP_S	Unimplemented Instruction

9.2 ARC Instructions

This section provides a complete definition for each ARCV2 instruction, presented in alphabetical order. The instruction name is given at the top left and top right of the page, along with a brief instruction description, and instruction type.

The following sub-headings are used in the description of each instruction.

Function	Summarizes the function of the instruction.
Extension Group	Indicates if instruction is in the baseline set or included only as part of an extension group
Operation	C style expression that describes the operation of the instruction. Where relevant, a block diagram of the computation may be shown.
Instruction Format	Summary of the instruction format.
Syntax Example	Single syntax example.
Flag Affected	List of status flags that are affected.
Description	Full description of the instruction.
Pseudo Code	Operation of the instruction described in C style pseudo code.
Assembly Code Example	Assembly coding example.
Syntax and Encoding	The syntax of the instruction and corresponding instruction encoding. Table 6-17 on page 270 lists the instruction syntax convention. Table 6-5 on page 262 and Table 6-6 on page 263 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ABS<.f>	b, c	00100bbb00101111FBBBCCCCC001001
ABS<.f>	b, u6	00100bbb01101111FBBBuuuuuu001001
ABS_S	b, c	01111bbbccc10001

ADC

Function

Addition with Carry

Extension Group

BASELINE

Operation

if (cc) $a = b + c + \text{carry}$

Instruction Format

op a, b, c

Syntax Example

ADC<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Add source operand 1 (b) and source operand 2 (c) and carry, and place the result in the destination register, a. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* ADC */
    dest = src1 + src2 + C_flag
    if F==1 then
        Z_flag = if dest==0 then 1 else
0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
ADC r1,r2,r3    ; Add r2 to r3 with carry and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADC<.f>	a,b,c	00100bbb00000001FBBBCCCCCAAAAAA
ADC<.f>	a,b,u6	00100bbb010000001FBBBuuuuuuAAAAAA
ADC<.f>	b,b,s12	00100bbb10000001FBBBssssssSSSSSS
ADC<.cc><.f>	b,b,c	00100bbb11000001FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ADD

Extension Group

BASELINE

Operation

if (cc) $a = b + c$

Instruction Format

op a, b, c

Syntax Example

ADD<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Add source operand 1 (b) to source operand 2 (c) and place the result in the destination register, a. Any flag updates occur only if the set flags suffix (.F) is used.



Note

For 16-bit encoded instructions that work on the stack pointer (SP) or global pointer (GP), the offset is aligned to 32-bit. For example, ADD_S sp, sp, u7 only needs to encode the top 5 bits because the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

Pseudo Code

```

if cc==true then                                /* ADD */
    dest = src1 + src2
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```


Assembly Code Example

```
ADD r1,r2,r3    ; Add contents of r2 with r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ADD<.f>	a,b,c	00100bbb0000000FBBBCCCCCAAAAAA
ADD<.f>	a,b,u6	00100bbb01000000FBBBuuuuuuAAAAAA
ADD<.f>	b,b,s12	00100bbb10000000FBBBssssssSSSSSS
ADD<.cc><.f>	b,b,c	00100bbb11000000FBBBCCCCC0QQQQQ
ADD<.cc><.f>	b,b,u6	00100bbb11000000FBBBuuuuuu1QQQQQ
ADD_S	a,b,c	01100bbbccc11aaa
ADD_S	b,b,h	01110bbbhhh000HH
ADD_S	h,h,s3	01110ssshhh001HH
ADD_S	b,b,limm	01110bbb11000011
ADD_S	0,limm,s3	01110sss11000111
ADD_S	b,sp,u7	11000bbb100uuuuu
ADD_S	b,b,u7	11100bbb0uuuuuuu
ADD_S	c,b,u3	01101bbbccc00uuu
ADD_S	SP,SP,u7	11000000101uuuuu
ADD_S	R0,GP,s11	1100111sssssssss

Encodings supported by the CODE DENSITY option

ADD_S	R0,b,u6	01001bbb0UUU1uuu
ADD_S	R1,b,u6	01001bbb1UUU1uuu

ADD1

Function

Addition with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b + (c \ll 1)$;

Instruction Format

op a, b, c

Syntax Example

ADD1<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 1. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD1 */
if cc==true then
  shiftedsrc2 = src2 << 1
  dest = src1 + shiftedsrc2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
    ( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31] )
```

Assembly Code Example

```
ADD1 r1,r2,r3    ; Add contents of r3 shifted left one bit to r2 and
                  ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADD1<.f>	a,b,c	00100bbb00010100FBBBCCCCCAAAAAA
ADD1<.f>	a,b,u6	00100bbb01010100FBBBuuuuuuAAAAAA
ADD1<.f>	b,b,s12	00100bbb10010100FBBBssssssSSSSSS
ADD1<.cc><.f>	b,b,c	00100bbb11010100FBBBCCCCC0QQQQQ
ADD1<.cc><.f>	b,b,u6	00100bbb11010100FBBBuuuuuu1QQQQQ
ADD1_S	b,b,c	01111bbbccc10100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ADD2

Function

Addition with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b + (c \ll 2)$;

Instruction Format

op a, b, c

Syntax Example

ADD2<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 2. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD2 */
if cc==true then
  shiftedsrc2 = (src2 << 2)
  dest = src1 + shiftedsrc2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
    ( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31] )
```

Assembly Code Example

```
ADD2 r1,r2,r3      ; Add contents of r3 shifted left two bits to r2 and
                   ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADD2<.f>	a,b,c	00100bbb00010101FBBBCCCCCAAAAAA
ADD2<.f>	a,b,u6	00100bbb01010101FBBBuuuuuuAAAAAA
ADD2<.f>	b,b,s12	00100bbb10010101FBBBssssssSSSSSS
ADD2<.cc><.f>	b,b,c	00100bbb11010101FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ADD3

Function

Addition with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b + (c \ll 3)$;

Instruction Format

op a, b, c

Syntax Example

ADD3<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 3. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD3 */
if cc==true then
  shiftedsrc2 = src2 << 3
  dest = src1 + shiftedsrc2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
    ( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31] )
```

Assembly Code Example

```
ADD3 r1,r2,r3    ; Add contents of r3 shifted left three bits to r2 and
                  ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ADD3<.f>	a,b,c	00100bbb00010110FBBBCCCCCAAAAAA
ADD3<.f>	a,b,u6	00100bbb01010110FBBBuuuuuuAAAAAA
ADD3<.f>	b,b,s12	00100bbb10010110FBBBssssssSSSSSS
ADD3<.cc><.f>	b,b,c	00100bbb11010110FBBBCCCCC0QQQQQ
ADD3<.cc><.f>	b,b,u6	00100bbb11010110FBBBuuuuuu1QQQQQ
ADD3_S	b,b,c	01111bbbccc10110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

AEX

Function

Swap contents of an auxiliary register with a core register.

Extension Group

BASELINE

Instruction Format

op b, [c]

Syntax Example

AEX<.cc> b, [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Swaps contents of an auxiliary register with a core register. This instruction is used by the micro-operations sequencer within the exception/interrupt entry/exit sequences. The b register operand of AEX cannot specify either a LImm data value or a null operand. Any use of LImm or null operand for the b register raises an [Illegal Instruction](#) exception.

Semantically, an AEX instruction behaves as the union of the LR and SR instructions, with the exception that an AEX can be conditional whereas LR/SR are not.

Pseudo Code

```
if cc==true then                                /* AEX */
    tmp = AuxRead (src2)
    AuxWrite (src2, b)
    b = tmp
```

Assembly Code Example

```
AEX R28, [AUX_USER_SP]    ; swap user and kernel stack pointers
```


Syntax and Encoding

For more information about the encodings, see “Key for 32-bit Addressing Modes and Encoding Conventions” and “Key for 16-bit Addressing Modes and Encoding Conventions” .

Instruction Code		
AEX	b, [c]	00100bbb00100111RBBBCCCCCRRRRRR
AEX	b, [u6]	00100bbb01100111RBBBuuuuuuRRRRRR
AEX	b, [s12]	00100bbb10100111RBBBssssssSSSSSS
AEX<.cc>	b, [c]	00100bbb11100111RBBBCCCCC0QQQQQ
AEX<.cc>	b, [u6]	00100bbb11100111RBBBuuuuuu1QQQQQ

AND

Function

Bitwise AND Operation

Extension Group

BASELINE

Operation

if (cc) $a = b \& c$;

Instruction Format

op a, b, c

Syntax Example

AND<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise AND of source operand 1 (b) with source operand 2 (c) with the result written to the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                /* AND */
    dest = src1 AND src2
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
```

Assembly Code Example

```
AND r1,r2,r3    ; AND contents of r2 with r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
AND<.f>	a, b, c	00100bbb00000100FBBBCCCCCAAAAAA
AND<.f>	a, b, u6	00100bbb01000100FBBBuuuuuuAAAAAA
AND<.f>	b, b, s12	00100bbb10000100FBBBssssssSSSSSS
AND<.cc><.f>	b, b, c	00100bbb11000100FBBBCCCCC0QQQQQ
AND<.cc><.f>	b, b, u6	00100bbb11000100FBBBuuuuuu1QQQQQ
AND_S	b, b, c	01111bbbccc00100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ASL

Function

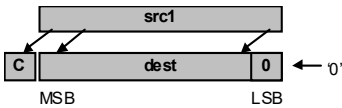
Arithmetic Shift Left

Extension Group

BASELINE

Operation

$$b = (\text{signed})\ c \ll 1;$$



Instruction Format

op b, c

Syntax Example

ASL<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if the sign bit changes after a shift

Description

Arithmetically, left shift the source operand (c) by one and place the result into the destination register (b). An ASL operation is effectively accomplished by adding the source operand upon itself (c + c), with the result being written into the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src + src                                /* ASL */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
ASL r1,r2    ; Arithmetic shift left contents of r2 by one bit and
              ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASL<.f>	b, c	00100bbb00101111FBBBCCCCC000000
ASL<.f>	b, u6	00100bbb01101111FBBBuuuuuu000000
ASL_S	b, c	01111bbbccc11011

ASL Multiple

Function

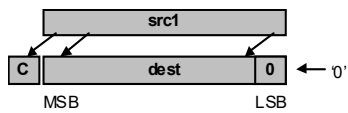
Multiple Arithmetic Shift Left

Extension Group

SHIFT_OPTION 2 or 3

Operation

if (cc) a = (signed) b << c;



Instruction Format

op a, b, c

Syntax Example

ASL<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Arithmetically, shift left b by c places and place the result in the destination register. Only the bottom 5 bits of c are used as the shift value. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* ASL */
  dest = src1 << (src2 & 31)                    /* Multiple */
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = if src2==0 then 0 else src1[32-src2]

```

Assembly Code Example

```

ASL r1,r2,r3    ; Arithmetic shift left contents of r2 by r3 bits
                 ; and write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASL<.f>	a,b,c	00101bbb0000000FBBBCCCCCAAAAAA
ASL<.f>	a,b,u6	00101bbb01000000FBBBuuuuuuAAAAAA
ASL<.f>	b,b,s12	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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ASR

Function

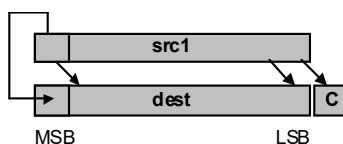
Arithmetic Shift Right

Extension Group

BASELINE

Operation

$b = (\text{signed})\ c \gg 1;$



Instruction Format

op b, c

Syntax Example

ASR<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Arithmetically right shift the source operand (c) by one and place the result into the destination register (b). The sign of the source operand is retained in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 1                                /* ASR */
if src[31]==1 then dest[31] = 1
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[0]
```

Assembly Code Example

```
ASR r1,r2      ; Arithmetic shift right contents of r2 by one bit and
                ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASR<.f>	b,c	00100bbb00101111FBBBCCCCC000001
ASR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000001
ASR_S	b,c	01111bbbccc11100

ASR multiple

Function

Multiple Arithmetic Shift Right

Extension Group

SHIFT_OPTION 2 or 3

Operation

if (cc) a = (signed) b >> c;



Instruction Format

op a, b, c

Syntax Example

ASR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Arithmetically, shift right b by c places and place the result in the destination register. Only the bottom 5 bits of c are used as the shift value. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* ASR */
  dest = ((signed)src1) >> (src2 & 31)              /* Multiple */
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = if src2==0 then 0 else src1[src2-1]

```

Assembly Code Example

```

ASR r1,r2,r3    ; Arithmetic shift right contents of r2 by r3 bits
                ; and write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASR<.f>	a,b,c	00101bbb0000010FBBBCCCCCAAAAAA
ASR<.f>	a,b,u6	00101bbb01000010FBBBuuuuuuAAAAAA
ASR<.f>	b,b,s12	00101bbb1000010FBBBssssssSSSSSS
ASR<.cc><.f>	b,b,c	00101bbb11000010FBBBCCCCC0QQQQQ
ASR<.cc><.f>	b,b,u6	00101bbb11000010FBBBuuuuuu1QQQQQ
ASR_S	c,b,u3	01101bbbccc11uuu
ASR_S	b,b,c	01111bbbccc11010
ASR_S	b,b,u5	10111bbb010uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ASR16

Function

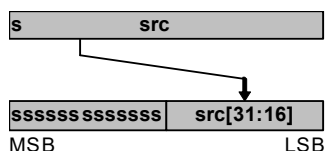
Arithmetic Shift Right 16

Extension Group

SHIFT_OPTION 1 or 3

Operation

$b = c \gg 16;$



Instruction Format

op b,c

Syntax Example

ASR16<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 16 places to the right. Set the upper 16 bits to 0xFFFF if the source operand is negative. Otherwise, clear the upper 16 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 16                                /* ASR16 */
if F==1 then
  Z_flag = if dest==0 then 1 else
0
  N_flag = dest[31]
```

Assembly Code Example

```
ASR16 r1,r2    ; Arithmetic shift of r2 16 places to the
                ; right, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASR16<.f>	b,c	00101bbb00101111FBBBCCCCC001100
ASR16<.f>	b,u6	00101bbb01101111FBBBuuuuuu001100

ASR8

Function

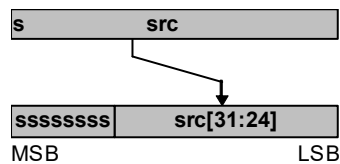
Arithmetic Shift Right 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$b = c \gg 8;$



Instruction Format

op b,c

Syntax Example

ASR8<.f> b,c

STATUS32 Flags Affected

Z	☒	= Set if result is zero
N	☒	= Set if most-significant bit of result is set
C	☐	= Unchanged
V	☐	= Unchanged

Description

Shift the source operand 8 places to the right. Set the upper 8 bits to 0xFF if the source operand is negative. Otherwise, clear the upper 8 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 8                                /* ASR8 */
if F==1 then
  Z_flag = if dest==0 then 1 else
0
  N_flag = dest[31]
```

Assembly Code Example

```
ASR8 r1,r2    ; Arithmetic shift of r2 8 places to the right, placing
               result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ASR8<.f>	b,c	00101 bbb 00 101111 FBBB CCCCC 001101
ASR8<.f>	b,u6	00101 bbb 01 101111 FBBB uuuuuu 001101

B

Function

Branch unconditionally

Extension Group

BASELINE

Operation

$PC = (PCL + s25);$

Instruction Format

op s25

Syntax Example

B<.d> s25

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

When an unconditional branch is used, the program execution is resumed at location PC (actually PCL) + relative displacement, where PC is the address of the B instruction. The unconditional branch far format has a maximum branch range of +/- 16 M. Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction. The status flags are not updated with this instruction.

[Table 6-37](#) on page 291 describes the delay slot modes, .d.

[Table 6-7](#) on page 264 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 B instruction with a delay slot must not include a long immediate operand. If included, the processor raises an Illegal Instruction error.

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```
if N=1 then                                /* B */
    DelaySlot(nPC)
    PC = cPC + rd
```

Assembly Code Example

```
B label          ; Branch to label
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
B<.d>          s25  00000 ssssssssss1SSSSSSSSSSNRtttt
```

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.
- The B_S instruction with a delay slot must not include a long immediate operand. If included, the processor raises an Illegal Instruction error.

Pseudo Code

```
KillDelaySlot(nPC)           /* B_S */
PC = cPCL + rd
```

Assembly Code Example

```
B_S label                    ; Branch to label
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
B_S      s10  1111000ssssssssss
```

BBIT0

Function

Branch on Bit Test Clear

Extension Group

BASELINE

Operation

if $((b \text{ AND } (1 < c)) == 0)$ 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 6-37](#) on page [291](#) lists the delay slot modes, <.d>.



Caution

- The BBIT0 instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction.
- A BBIT0 instruction with a delay slot must not include a long immediate operand. If included, the processor raises an Illegal Instruction error.

Some implementations of the ARCV2 may predict the outcome of BBIT0 instructions. These predictions may be static, that is, based on the instruction pattern, whereas others may be dynamic, that is, based on the observed execution history of the branch. A description of the static prediction semantics for BBIT0 instructions can be found in section [Table 6-10](#) on page 266, and more details can be found in the section [“Branch On Compare or Bit Test”](#).

Pseudo Code

```
if (src1 & (1 << (src2 & 31)))==0    /* BBIT0 */
then
  if N=1 then
    DelaySlot(nPC)
    KillDelaySlot(nPC + 1)
    PC = PCL + s9
  else
    PC = nPC
```

Assembly Code Example

```
BBIT0 r1,9,label      ; Branch to label if bit 9
                       ; of r1 is clear
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BBIT0<.d><.T>	b,c,s9	00001bbbssssssss1SBBBCCCCCN0Y110
BBIT0<.d><.T>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN1Y110

BBIT1

Function

Branch on Bit Test Set

Extension Group

BASELINE

Operation

if ((b AND (1<<c)) == 1) PC = PCL+s9;

Instruction Format

op b, c, s9

Syntax Example

BBIT1<.d><.T> b, c, s9

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Test a bit within source operand 1 (b) to see if the bit is set (1). Source operand 2 (c) explicitly specifies the position of the bit that is to be tested within source operand 1 (b).

Only the bottom 5 bits of operand 2 are used as the bit position.

If the selected bit is set, the branch is taken. The branch target is computed as the sum of the current 32-bit word-aligned PC and the signed 16-bit half-word displacement value specified by the 9-bit literal source operand (s9).

Because the execution of the instruction that is in the delay slot is controlled by the delay slot mode, the instruction must never be the target of any branch or jump instruction. The status flags are not updated by this instruction.

[Table 6-37](#) on page [291](#) lists the delay slot modes, <.d>.

**Caution**

- The BBIT1 instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction.
- A BBIT1 instruction with a delay slot must not include a long immediate operand. If included, the processor raises an Illegal Instruction error.

Some implementations of the ARCV2 may predict the outcome of BBIT1 instructions. These predictions may be static, that is, based on the instruction pattern, whereas others may be dynamic, that is, based on the observed execution history of the branch. A description of the static prediction semantics for BBIT instructions can be found in section [“Static Branch Prediction Mode”](#), and more details can be found in section [“Branch On Compare or Bit Test”](#).

Pseudo Code

```
if (src1 & (1 << (src2 & 31))) != 0 then          /* BBIT1 */
  if N=1 then
    DelaySlot(nPC)
    KillDelaySlot(dPC)
    PC = cPCL + rd
  else
    PC = nPC
```

Assembly Code Example

```
BBIT1 r1,9,label    ; Branch to label if bit 9 of r1 is set
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BBIT1<.d><.T>	b, c, s9	00001bbbsssssss1SBBBCCCCCN0Y111
BBIT1<.d><.T>	b, u6, s9	00001bbbsssssss1SBBBuuuuuN1Y111

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 6-37](#) on page 291.

The condition codes, cc, are described in [Table 6-7](#) on page 264.

**Caution**

- The Bcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.
- The Bcc instruction with a delay slot must not include a long immediate operand. If included, the processor raises an Illegal Instruction error.

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```
if cc==true then                                /* B */
  if N=1 then
    DelaySlot(nPC)
    PC = cPC + rd
  else
    PC = nPC
```

Assembly Code Example

```
BEQ label      ; Branch to label if Z flag is set Branch to label and
                ; execute the instruction in the delay ; slot if N flag
BPL.D label    ; is clear
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

B<cc><.d> s21 00000 **ssssssssss**0**ssssssssss**N**QQQQQ**

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 6-7](#) on page [264](#) 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.
- The Bcc_S instruction with a delay slot must not include a long immediate operand. If included, the processor raises an Illegal Instruction error.

Pseudo Code

```

if cc==true then                                /* Bcc_S */
    KillDelaySlot(nPC)
    PC = cPCL + rd
else
    PC = nPC

```

Assembly Code Example

```

BEQ_S label    ; Branch to label if Z flag is set
BNE_S label    ; Branch to label if N flag is clear

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BEQ_S	s10	11110 01 ssssssss
BNE_S	s10	11110 10 ssssssss
BGT_S	s7	1111011 000 ssssss
BGE_S	s7	1111011 001 ssssss
BLT_S	s7	1111011 010 ssssss
BLE_S	s7	1111011 011 ssssss
BHI_S	s7	1111011 100 ssssss
BHS_S	s7	1111011 101 ssssss
BLO_S	s7	1111011 110 ssssss
BLS_S	s7	1111011 111 ssssss

BCLR

Function

Bit Clear

Extension Group

BASELINE

Operation

if (cc) then $a = (b \& \sim(1 \ll c));$

Instruction Format

op a, b, c

Syntax Example

BCLR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Clear (0) an individual bit within the value that is specified by the source operand 1 (b). The source operand 2 (c) contains a value that explicitly defines the bit-position that is to be cleared in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The result is written into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* BCLR */
  dest = src1 AND NOT(1 << (src2 & 31))
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]

```

Assembly Code Example

```
BCLR r1,r2,r3    ; Clear bit r3 of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BCLR<.f>	a,b,c	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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BI

Function

Branch Indexed 32-bit Full-Word Table

Extension Group

CODE_DENSITY

Operation

$$PC = next\_PC + (c \ll 2)$$

Instruction Format

op c

Syntax Example

BI [r1]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The relative branch offset is defined by shifting the contents of the (c) operand register left by 2 places. Program execution resumes at the location given by the next sequential PC plus the branch offset.

Pseudo Code

```
PC = next_PC + (c << 2)          /* BI */
```

Assembly Code Example

```
BI [r1]      ;   branch to next_PC+r1<<2
```


Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
BI    [c]    00100RRR001001000RRRCCCCCRRRRRR
```

BIC

Function

Bitwise AND Operation with Inverted Source

Extension Group

BASELINE

Operation

if (cc) $a = b \& \sim c$;

Instruction Format

op a, b, c

Syntax Example

BIC<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise AND of source operand 1 (b) with the inverse of source operand 2 (c) with the result written to the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                     /* BIC */
  dest = src1 AND NOT src2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
BIC r1,r2,r3    ; AND r2 with the NOT of r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BIC<.f>	a, b, c	00100bbb00000110FBBBCCCCCAAAAA
BIC<.f>	a, b, u6	00100bbb01000110FBBBuuuuuuAAAAA
BIC<.f>	b, b, s12	00100bbb10000110FBBBssssssSSSSSS
BIC<.cc><.f>	b, b, c	00100bbb11000110FBBBCCCCC0QQQQQ
BIC<.cc><.f>	b, b, u6	00100bbb11000110FBBBuuuuuu1QQQQQ
BIC_S	b, b, c	01111bbbccc00110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BIH

Function

Branch Indexed to Half-Word Table

Extension Group

CODE_DENSITY

Operation

$$PC = next\_PC + (c \ll 1)$$

Instruction Format

op c

Syntax Example

BIH [r1]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The relative branch offset is defined by shifting the contents of the (c) operand register to left by one place. Program execution resumes at the location given by the next sequential PC + the branch offset.

Pseudo Code

```
PC = next_PC + (c << 1)          /* BIH */
```

Assembly Code Example

```
BIH [r1]    ; branch to next_PC + r1 << 1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BIH [c] 00100RRR001001010RRRCCCCRRRRRR

BLcc

Function

Branch and Link

Extension Group

BASELINE

Operation

if (cc) {BLINK = NEXT_PC; PC = PCL +s21;}

Instruction Format

op s21

Syntax Example

BL<.cc><.d> s21

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

When a conditional branch and link is used and the specified condition is met (cc), program execution is resumed at the PCL (the PC, 32-bit aligned) plus a relative displacement, where PC is the address of the BLcc instruction. At the same time, the return address is stored in the link register BLINK (R31). This address is taken either from the first instruction following the branch (current PC) or the instruction after that (next PC) according to the delay slot mode (.d).

The delay slot modes, .d, are described in [Table 6-37](#) on page 291.

The condition codes, cc, are described in [Table 6-7](#) on page 264.



Caution

The BLcc and BL_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The conditional branch and link instruction has a maximum branch range of ± 1 M. The unconditional branch far format has a maximum branch range of ± 16 M. The target address for any branch and link instruction must be 32-bit aligned.

Because the execution of the instruction that is in the delay slot is controlled by the delay slot mode, the instruction must never be the target of any branch or jump instruction. The status flags are not updated with this instruction.

**Note**

The 16-bit-encoded instructions have a target address aligned to 32 bits. For this reason, a special encoding allows for a larger branch displacement. For example, BL_S s13 only needs to encode the top 11 bits because the bottom 2 bits of s13 are always zero because of the 32-bit data alignment.

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- ❑ Another jump or branch instruction
- ❑ Conditional loop instruction (LPcc)
- ❑ Return from interrupt ([RTIE](#))
- ❑ Any instruction with long-immediate data as a source operand

Pseudo Code

```
if cc==true then    /* BLcc */
  if N=1 then       /*nPC = instruction after BLcc
    BLINK = dPC      /*dPC= instruction after delay slot
    DelaySlot(nPC)   /*cPCL=BLcc instruction address, 32-bit aligned (PCL)
  else              /*N: delay slot mode (.d)
    BLINK = nPC
    PC = cPCL + rd
  else
    PC = nPC
```

Assembly Code Example

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

```
BLEQ label        ; if the Z flag is set then branch and link to label and
                   ;store the return address in BLINK
```

Syntax and Encoding

Instruction Code			
BL<.cc><.d>	s21	00001	ssssssssss00SSSSSSSSSSNQQQQQ
BL<.d>	s25	00001	ssssssssss10SSSSSSSSSSNRtttt
BL_S	s13	11111	ssssssssssss

BMSK

Function

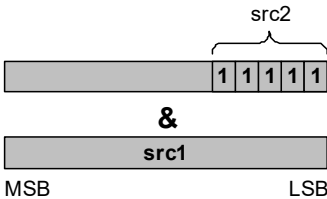
Bit Mask

Extension Group

BASELINE

Operation

if (cc) then a = (b & ((1 << (c+1))-1);



Instruction Format

op a, b, c

Syntax Example

BMSK<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Source operand 2 (c) specifies the size of a 32-bit mask value in terms of logical 1's starting from the LSB of a 32-bit register up to and including the bit specified by operand 2 (c). Only the bottom 5 bits of src2 are used as the bit value.

A logical AND is performed with the mask value and source operand (b). The result is written into the destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* BMSK */
    dest = src1 AND ((1 << ((src2 &
31)+1))-1)
    if F==1 then
        Z_flag = if dest==0 then 1 else
0
        N_flag = dest[31]

```

Assembly Code Example

```

BMSK r1,r2,31      ; Do not mask any bits of r2, write result into r1
BMSK r1,r2,7       ; Mask out the top 24 bits of r2 and write result into r1
BMSK r1,r2,0       ; Mask out all except bit 0 of r2 and write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BMSK<.f>	a,b,c	00100bbb00010011FBBBCCCCC
BMSK<.f>	a,b,u6	00100bbb01010011FBBBuuuuuu
BMSK<.f>	b,b,s12	00100bbb10010011FBBBssssss
BMSK<.cc><.f>	b,b,c	00100bbb11010011FBBBCCCCC
BMSK<.cc><.f>	b,b,u6	00100bbb11010011FBBBuuuuuu
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BMSKN

Function

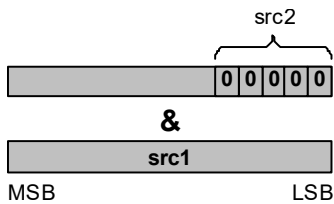
Bit Mask Negated

Extension Group

BASELINE

Operation

if (cc) then a = (b & (~(1 << (c+1))-1));



Instruction Format

op a, b, c

Syntax Example

BMSKN <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Source operand 2 (c) specifies the size of a 32-bit mask value in terms of logical 0's starting from the LSB of a 32-bit register up to and including the bit specified by operand 2(c). All bits beyond that position in the mask are 1's. Only the bottom 5 bits of src2 are used to define the size of the zero portion of the bit mask.

A logical AND is performed on the mask value and source operand (b). The result is written into the destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /*BMSKN */
    dest = src1 AND ~(1 << ((src2 & 31)+1))-1)
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]

```

Assembly Code Example

```
BMSKN r1,r2,8 ; r1 is set to (r2 & 0xFFFFFE0)
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BMSKN<.f>	a,b,c	00100bbb00101100FBBBCCCCCAAAAAA
BMSKN<.f>	a,b,u6	00100bbb01101100FBBBuuuuuuAAAAAA
BMSKN<.f>	b,b,s12	00100bbb10101100FBBBssssssSSSSSS
BMSKN<.cc><.f>	b,b,c	00100bbb11101100FBBBCCCCC0QQQQQ
BMSKN<.cc><.f>	b,b,u6	00100bbb11101100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BRcc

Function

Compare and Branch

Extension Group

BASELINE for 32-bit BRcc operations

Operation

if (cc(b,c)) PC = PCL+s9

Instruction Format

op b, c, s9

Syntax Example

BREQ<.d><.T> b, c, s9

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

The BRcc instruction performs one of six possible 32-bit relational tests on its operands, and if the result is true, the branch is taken. If a delay-slot is specified, using delay slot mode shown in [Table 6-37](#) on page 291, one instruction immediately following the branch in program order is executed before the branch takes effect.

The branch target is computed as the sum of the branch’s 32-bit word-aligned PC plus the 16-bit half-word aligned displacement value given by the signed 9-bit literal operand (rd).

The available branch conditions are shown in the following table:

Instruction	Description	Branch Condition
BREQ	Branch if Equal	if (b==c) PC = (PCL + s9)
BRNE	Branch if Not Equal	if (b!=c) PC = (PCL + s9)
BRLT	Branch if Less Than (Signed)	if (b<c) PC = (PCL + s9)

BRGE	Branch if Greater Than or Equal (Signed)	if (b>=c) PC = (PCL + s9)
BRLO	Branch if Lower Than (Unsigned)	if (b<c) PC = (PCL + s9)
BRHS	Branch if Higher Than or Same (Unsigned)	if (b>=c) PC = (PCL + s9)

For 16-bit compare and branch instructions, BRNE_S compares the source operand 1 (b) with 0, and if src1 is not equal to zero, the branch is taken.

BREQ_S performs the same comparison. However, the branch is taken when the source operand 1 (b) is equal to zero.

Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction.

The status flags are not updated by this instruction.



Caution

The BRcc and BRcc_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

Some implementations of the ARCV2 may predict the outcome of BRcc instructions. These predictions may be static, that is, based on the instruction pattern, whereas others may be dynamic, that is, based on the observed execution history of the branch. A description of the static prediction semantics for BRcc instructions can be found in section [“Static Branch Prediction Mode”](#), and more details can be found in section [“Branch On Compare or Bit Test”](#). The interpretation of the <T> syntax can be found in [Table 6-10](#) on page 266.

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BREQ <.d><.T>	b, c, s9	00001bbbssssss1SBBBCCCCCN0Y000
BREQ <.d><.T>	b, u6, s9	00001bbbssssss1SBBBuuuuuuN1Y000
BRNE <.d><.T>	b, c, s9	00001bbbssssss1SBBBCCCCCN0Y001
BRNE <.d><.T>	b, u6, s9	00001bbbssssss1SBBBuuuuuuN1Y001
BRLT <.d><.T>	b, c, s9	00001bbbssssss1SBBBCCCCCN0Y010
BRLT <.d><.T>	b, u6, s9	00001bbbssssss1SBBBuuuuuuN1Y010
BRGE <.d><.T>	b, c, s9	00001bbbssssss1SBBBCCCCCN0Y011
BRGE <.d><.T>	b, u6, s9	00001bbbssssss1SBBBuuuuuuN1Y011

BRLO <.d><.T>	b, c, s9	00001bbbssssssss1SBBBCCCCCN0Y100
BRLO <.d><.T>	b, u6, s9	00001bbbssssssss1SBBBuuuuuuN1Y100
BRHS <.d><.T>	b, c, s9	00001bbbssssssss1SBBBCCCCCN0Y101
BRHS <.d><.T>	b, u6, s9	00001bbbssssssss1SBBBuuuuuuN1Y101
BRNE_S	b, 0, s8	11101bbb1ssssssss
BREQ_S	b, 0, s8	11101bbb0ssssssss

Additional Pseudo Syntax and Encoding

The following pseudo-instructions for missing conditions are available using existing encodings:

BRGT<.d> b, c, s9	Encode as BRLT<.d> c, b, s9
BRGT<.d> b, u6, s9	Encode as BRGE<.d> b, u6+1, s9
BRHI<.d> b, c, s9	Encode as BRLO<.d> c, b, s9
BRHI<.d> b, u6, s9	Encode as BRHS<.d> b, u6+1, s9
BRLE<.d> b, c, s9	Encode as BRGE<.d> c, b, s9
BRLE<.d> b, u6, s9	Encode as BRLT<.d> b, u6+1, s9
BRLS<.d> b, c, s9	Encode as BRHS<.d> c, b, s9
BRLS<.d> b, u6, s9	Encode as BRLO b, u6+1, s9

BRK

Function

Breakpoint

Extension Group

BASELINE

Operation

Halt and flush the processor

Instruction Format

op

Syntax Example

BRK_S

Flag Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
BH	•	= 1
H	•	= 1

Description

The breakpoint instruction halts the processor without advancing the program counter or otherwise modifying the state of the processor or its memory.

The BRK_S instruction is normally used by an external debug host to set breakpoints during debugging. The debugger typically replaces the instruction at the breakpoint address with the BRK_S instruction to force the processor to halt whenever PC reaches the breakpoint address. To restart the processor after taking a breakpoint, the debugger must write the original instruction back to memory and issue an instruction cache invalidation command. The external debug host can then restart the processor, and resume normal execution of the original instruction from the breakpoint address.

There is no limit to the number of breakpoints that can be inserted into a piece of code.



The breakpoint instruction sets the BH bit in the Debug register, which allows the debugger to determine what caused the ARCV2-based processor to halt. The BH bit is cleared when the Halt bit in the Status register is cleared, for example, by restarting or single-stepping the ARCV2-based processor.

The breakpoint instruction is a kernel-only instruction unless enabled by the UB bit in the DEBUG register. Any attempt to execute BRK_S, with the UB bit set to 0, raises a privilege violation exception (see [Machine Check, Internal Instruction Memory Error](#)).

The ARCV2 provides both a 32-bit (BRK) and a 16-bit (BRK_S) encoding of the breakpoint instruction.

The breakpoint instruction can be placed anywhere in a program, including the delay slot of branch or jump instructions, and also immediately following a BRcc, a BBIT0, or a BBIT1 instruction.

Pseudo Code

```
FlushPipe()                      /* BRK_S, BRK */
DEBUG[BH] = 1
Halt()
```

Assembly Code Example

A breakpoint instruction may be inserted into any position.

```
MOV    r0, 0x04
ADD    r1, r0, r0
XOR.F  0, r1, 0x8
BRK_S                                     ; ←--- break here
SUB    r2, r0, 0x3
ADD.NZr1, r0, r0
JZ.D   [r8]
OR     r5, r4, 0x10
```

Breakpoints may be inserted at a jump delay slot.

```
MOV    r0, 0x04
ADD    r1, r0, r0
XOR.F  0, r1, 0x8
SUB    r2, r0, 0x3
ADD.NZr1, r0, r0
JZ.D   [r8]
BRK_S                                     ; ←--- break here
RLC    r2, r1
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code

BRK 00100**101**01**101111**0**000**RRRRRR**111111**

BRK_S 0111111111111111



Note

By software convention, the reserved fields. *b* and *c* are set to 0x7 for the BRK_S instruction (the encoding is 0x7FFF).

BSET

Function

Bit Set

Extension Group

BASELINE

Operation

if (cc) a = (b | (1<<c));

Instruction Format

op a, b, c

Syntax Example

BSET<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Set (to 1) an individual bit within the value that is specified by source operand 1 (b). Source operand 2 (c) contains a value that explicitly defines the bit-position that is to be set in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The result is written into the destination register (dest). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* BSET */
    dest = src1 OR (1 << (src2 & 31))
    if F==1 then
        Z_flag = 0
        N_flag = dest[31]

```

Assembly Code Example

```
BSET r1,r2,r3    ; Set bit r3 of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BSET<.f>	a,b,c	00100bbb00001111FBBBCCCCCAAAAA
BSET<.f>	a,b,u6	00100bbb01001111FBBBuuuuuuAAAAA
BSET<.f>	b,b,s12	00100bbb10001111FBBBssssssSSSSSS
BSET<.cc><.f>	b,b,c	00100bbb11001111FBBBCCCCC0QQQQQ
BSET<.cc><.f>	b,b,u6	00100bbb11001111FBBBuuuuuu1QQQQQ
BSET_S	b,b,u5	10111bbb100uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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 writeback.



Note

BTST and BTST_S always set the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                                /* BTST */
    alu = src1 AND (1 << (src2 & 31))
    Z_flag = if alu==0 then 1 else 0
    N_flag = alu[31]
```

Assembly Code Example

```
BTST r2,28    ; Test bit 28 of r2 and update flags on result
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
BTST	b, c	00100bbb000100011BBBCCCCCRRRRRR
BTST	b, u6	00100bbb010100011BBBuuuuuuRRRRRR
BTST	b, s12	00100bbb100100011BBBssssssSSSSSS
BTST<.cc>	b, c	00100bbb110100011BBBCCCCC0QQQQQ
BTST<.cc>	b, u6	00100bbb110100011BBBuuuuuu1QQQQQ
BTST_S	b, u5	10111bbb111uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

BXOR

Function

Bit Exclusive OR (Bit Toggle)

Extension Group

BASELINE

Operation

if (cc) $a = (b \wedge (1 \ll c));$

Instruction Format

op a, b, c

Syntax Example

BXOR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logically XOR source operand 1 (b) with a bit mask specified by source operand 2 (c). Source operand 2 (c) explicitly defines the bit that is to be toggled in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The result is written to the destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                /* BXOR */
    dest = src1 XOR (1 << (src2 & 31))
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
```

Assembly Code Example

```
BXOR r1,r2,r3      ; Toggle bit r3 of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

BXOR<.f>	a, b, c	00100bbb00010010FBBBCCCCC	AAAAAA
BXOR<.f>	a, b, u6	00100bbb01010010FBBBuuuuuu	AAAAAA
BXOR<.f>	b, b, s12	00100bbb10010010FBBBssssss	SSSSSS
BXOR<.cc><.f>	b, b, c	00100bbb11010010FBBBCCCCC	0QQQQQ
BXOR<.cc><.f>	b, b, u6	00100bbb11010010FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

CLRI

Function

CLRI forces the STATUS32.IE bit to 0, disabling interrupts.

Extension Group

HAS_INTERRUPTS == 1

Operation

$\{\text{dest} \leftarrow \{ 26'd0, 1'b1, \text{STATUS32.IE}, \text{STATUS32.E}[3:0] \}; \text{STATUS32.IE} \leftarrow 0\}$

Instruction Format

op c

Syntax Example

CLRI c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

CLRI forces the STATUS32.IE bit to 0, disabling interrupts. If a destination register is specified as a c operand, this instruction saves the interrupt state from the STATUS32 register to the destination register before disabling interrupts.

The CLRI instruction is available only in kernel mode. Using this instruction in the user mode raises a [Privilege Violation, Kernel Only Access](#) exception. CLRI is a serializing instruction similar to the KFLAG instruction. This behavior means after this instruction commits, the pipeline is flushed.

Pseudo Code

```
{
    dst <- { 26'd0, 1'b1, STATUS32.IE, STATUS32.E[3:0] }
    STATUS32.IE <- 0
}
```

/* CLRI */

Assembly Code Example

```
CLRI R0      ;clear interrupts and capture interrupt state in R0
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
CLRI	c	00100111001011110000cccccc111111
CLRI	u6	00100111011011110000uuuuuu111111

If the destination register value is 62, or if a u6 operand is specified, CLRI does not save the interrupt state to the destination register value.

 **Note**

Use of limm operands with the CLRI instruction is deprecated. If you use a limm operand with the CLRI instruction, an Illegal Instruction exception is raised.

If a destination C register, whose encoded value is not 62, is specified, CLRI saves the interrupt state from the STATUS32 register to the destination register before disabling interrupts. The resulting destination register can be subsequently used as an operand to the SETI instruction to restore the interrupt enable and interrupt priority level to their values prior to execution of the CLRI instruction. The 32-bit destination register is assigned the following values:

dst [31:6]	dst [5]	dst[4]	dst[3:0]
26'd0	1	STATUS32.IE	STATUS32.E [3:0]

CMP

Function

Comparison

Extension Group

BASELINE

Operation

if (cc) $b - c$;

Instruction Format

op b, c

Syntax Example

CMP<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

A comparison is performed by subtracting source operand 2 (c) from source operand 1 (b) and subsequently updating the flags.



Note

The CMP instruction always sets the flags even though there is no associated flag setting suffix.

There is no destination register. Therefore, the result of the subtraction is discarded.

Pseudo Code

```

if cc==true then                                /* CMP */
    alu = src1 - src2
    Z_flag = if alu==0 then 1 else 0
    N_flag = alu[31]
    C_flag = Carry()
    V_flag = Overflow()

```

Assembly Code Example

```

CMP r1,r2    ; Subtract r2 from r1 and set the flags on the result

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

CMP and CMP_S always set the flags even though there is no associated flag setting suffix.

Instruction Code

CMP	b,c	00100bbb000011001BBBCCCCCRRRRRR
CMP	b,u6	00100bbb010011001BBBuuuuuuRRRRRR
CMP	b,s12	00100bbb100011001BBBssssssSSSSSS
CMP<.cc>	b,c	00100bbb110011001BBBCCCCC0QQQQQ
CMP<.cc>	b,u6	00100bbb110011001BBBuuuuuu1QQQQQ
CMP_S	b,h	01110bbbh100HH
CMP_S	h,s3	01110sssh101HH
CMP_S	b,u7	11100bbb1uuuuuu

DBNZ

Function

Decrement and Branch if Non-zero

Extension Group

BASELINE

Operation

if (src != 1) PC = PCL + s13; src = src - 1

Instruction Format

op b, s13

Syntax Example

DBNZ<.d> b, s13

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The DBNZ instruction decrements its source register operand (src), and if the result is non-zero it branches to the location defined by a signed half-word displacement operand (s13).

This instruction supports only the REG-S12 operand format, as it always requires one register and a signed 12-bit literal operand. Any use of other operand formats raises an Illegal Instruction exception. The register operand must be a writable general-purpose register, that is, it cannot be a long-immediate indicator (r62), nor can it be the PCL register (r63). Any attempt to use one of those register operands also raises an Illegal Instruction exception.

The 13-bit literal operand (s13) is encoded in the DBNZ instruction format as an s12 field in which the least-significant bit of the displacement has been omitted. This is because the displacement is always half-word aligned. The 13-bit relative displacement is added to the current 32-bit word-aligned PC value (PCL) to determine the branch target address.

When DBNZ is used to implement a loop-closing branch the target address represents the address of the first instruction of loop body.

The DBNZ instruction may optionally specify that a delay-slot instruction follows. The delay slot modes, .d, are described in [Table 6-37](#) on page 291.



Caution

The DBNZ instruction cannot appear in the delay slot of another branch or jump instruction, nor can it appear in the execution slot of an EI_S instruction.

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```

if (src != 1) {                                     /* DBNZ */
    if (Instruction[16])
        STATUS32.DE = 1;
        BTA = cPCL + offset
    } else {
        PC = cPCL + offset
    }
}
src = src - 1

```

Assembly Code Example

```

DBNZ r1, label          ; decrement r1, and branch to label if result
                        ; is non-zero

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```

DBNZ<.d>    b, s13    00100bbb1000110N0BBBssssssSSSSSS

```


DIV

Function

2's Complement Integer Divide. For the DSP version, see [DIV](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

if (cc) then $a = b / c$;

Instruction Format

op a, b, c

Syntax Example

DIV <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if divisor is zero or if an arithmetic overflow occurs

Description

If the divisor (c) is non-zero, the destination register is assigned the quotient obtained by signed integer division of source operand (b) by source operand (c).

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

An arithmetic overflow is signaled if the resulting value cannot be represented in 32 bits. This overflow occurs only in one specific case, which is the division of the largest representable negative value (0x80000000) by -1. If an arithmetic overflow occurs, the overflow flag (V) is set to 1.

If a division-by-zero is attempted, or if the result cannot be represented in 32 bits, when the EV_DivZero exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, “Implementation Behavior on Division Overflow”](#).

Pseudo Code

```
if (cc == true) {
    if ((src2 != 0) && ((src1 != 0x80000000) || src2 != 0xffffffff))
    {
        dest = src1 / src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = dest[31];
            V = 0;
        }
    } else {
        if ((src2 == 0) && (STATUS32.DZ == 1))
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1){
                V = 1;
            }
        }
    }
}
```

Assembly Code Example

```
DIV r1,r2,r3 ; r1 is assigned r2 / r3
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
DIV<.f>	a,b,c	00101bbb00000100FBBBCCCCCAAAAAA
DIV<.f>	a,b,u6	00101bbb01000100FBBBuuuuuuAAAAAA
DIV<.f>	b,b,s12	00101bbb10000100FBBBssssssSSSSSS
DIV<.cc><.f>	b,b,c	00101bbb11000100FBBBCCCCC0QQQQQ
DIV<.cc><.f>	b,b,u6	00101bbb11000100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DIVU

Function

Unsigned Integer Divide. For the DSP version, see [DIVU](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

if (cc) then $a = b / c$;

Instruction Format

op a, b, c

Syntax Example

DIVU <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always cleared
C		= Unchanged
V	•	= Set if divisor is zero

Description

If the divisor (c) is non-zero, the destination register is assigned the quotient obtained by unsigned integer division of source operand (b) by source operand (c).

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

If a division- by-zero is attempted, when the EV_DivZero exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, “Implementation Behavior on Division Overflow”](#).

If the flag-update bit is set, an unsigned DIVU operation always clears the N-flag.

Pseudo Code

```
if (cc == true) {
    if (src2 != 0) {
        dest = src1 / src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = 0;
            V = 0;
        }
    } else {
        if (STATUS32.DZ == 1)
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1){
                V = 1;
            }
        }
    }
}
```

Assembly Code Example

```
DIVU r1,r2,r3 ; r1 is assigned r2 / r3
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
DIVU<.f>	a,b,c	00101bbb00000101FBBBCCCCCAAAAAA
DIVU<.f>	a,b,u6	00101bbb01000101FBBBuuuuuuAAAAAA
DIVU<.f>	b,b,s12	00101bbb10000101FBBBssssssSSSSSS
DIVU<.cc><.f>	b,b,c	00101bbb11000101FBBBCCCCC0QQQQQ
DIVU<.cc><.f>	b,b,u6	00101bbb11000101FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DMACH

This section describes the standard DMACH instruction. For the DSP version, see [DMACH](#).

Function

Dual 16x16 signed integer multiply and accumulate.

Extension Group

MPY_OPTION == 7 or has_dsp==true.

Operation

```
if (cc) {
    result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
    a = result.w0;
    acc = result;
}
```

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) { /* DMACH */
    result = acc + (b.h0 * c.h0) + (b.h1 * c.h1);
    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” and “Key for 16-bit Addressing Modes and Encoding Conventions” .

Instruction Code		
DMACH<.f>	a,b,c	00101bbb00010010FBBBCCCCCAAAAAA
DMACH<.f>	a,b,u6	00101bbb01010010FBBBuuuuuuAAAAAA
DMACH<.f>	b,b,s12	00101bbb10010010FBBBssssssSSSSSS
DMACH<.cc><.f>	b,b,c	00101bbb11010010FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section “Dual-operand Instructions, F32_GEN4” .

DMACHU

This section describes the standard DMACHU instruction. For the DSP version, see [DMACHU](#).

Function

Dual unsigned integer 16x16 multiply and accumulate

Extension Groups

MPY_OPTION = or has_dsp==true.

Operation

```
if (cc) {
    result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
    a = result.w0;
    acc = result;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. w0 represents the lower 32-bits of an operand. For more information about 64-bit operands, see [“Data Formats”](#).

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

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
DMACHU<.f>	a,b,c	00101bbb00010011FBBBCCCCCAAAAAA
DMACHU<.f>	a,b,u6	00101bbb01010011FBBBuuuuuuAAAAAA
DMACHU<.f>	b,b,s12	00101bbb10010011FBBBsssssssSSSSSS
DMACHU<.cc><.f>	b,b,c	00101bbb11010011FBBBCCCCC0QQQQQ
DMACHU<.cc><.f>	b,b,u6	00101bbb11010011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DMACWH

Function

Dual 32x16 multiply and accumulate. For the DSP version, see [DMACWH](#).

Extension Group

MPY_OPTION == 9 or dsp_wide==true.

Operation

```
if (cc) {
    result = acc + ((B.w0 * c.h0) + (B.w1 * c.h1));
    A = result;
    acc = result;
}
```



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 and B are 64-bit operands, and must be specified as even numbered registers. For more information about 64-bit operands, see ["Data Formats"](#).

Instruction Format

op A, B, c

Syntax Example

DMACWH <.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 sum of the accumulator and the two 32x16 products formed by multiplying B.w0 with c.h0, and B.w1 with c.h1. The resulting sum is assigned to the destination register pair, and is also assigned to the 64-bit accumulator (ACCH, ACCL). In common with all operations that produce a 64-bit result, the 64-bit operands must be an even-numbered register (or the null destination).

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) {                                     /* DMACWH */
    result = acc + ((B.w0 * c.h0)+(B.w1 * c.h1));
    A = result;
    acc = result;
}
```

Assembly Code Example

```
DMACWH r0,r2,r5                               ; dual 32x16 MAC of r2 and r5
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
DMACWH<.f>	a,b,c	00101bbb00110110FBBBCCCCCAAAAAA
DMACWH<.f>	a,b,u6	00101bbb01110110FBBBuuuuuuAAAAAA
DMACWH<.f>	b,b,s12	00101bbb10110110FBBBsssssssSSSSSS
DMACWH<.cc><.f>	b,b,c	00101bbb11110110FBBBCCCCC0QQQQQ
DMACWH<.cc><.f>	b,b,u6	00101bbb11110110FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DMACWHU

Function

Dual unsigned 32x16 multiply and accumulate. For the DSP version, see [DMACWHU](#).

Extension Group

MPY_OPTION == 9 or dsp_wide==true.

Operation

```
if (cc) {
    result = acc + ((B.w0 * c.h0) + (B.w1 * c.h1));
    A = result;
    acc = result;}
```



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` and `B` are 64-bit operands, and must be specified as even numbered registers For more information about 64-bit operands, see [“Data Formats”](#).

Instruction Format

op A,B,c

Syntax Example

DMACWHU <.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

Compute the sum of the accumulator and the two unsigned 32x16 products formed by multiplying `B.w0` with `c.h0`, and `B.w1` with `c.h1`. The resulting sum is assigned to the destination register pair, and is also assigned to the 64-bit accumulator (ACCH, ACCL).

The `A` operand must be an even-numbered register (or the null destination).

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.w0 * c.h0)+(B.w1 * c.h1));
    A = result;
    acc = result;
}
```

/* DMACWHU */

Assembly Code Example

```
DMACWHU r0,r2,r4      ; dual unsigned 32x16 MAC of (r3, r2) and 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](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

26

Instruction Code		
DMACWHU<.f>	a,b,c	00101bbb00110111FBBBCCCCC AAAAAA
DMACWHU<.f>	a,b,u6	00101bbb01110111FBBBuuuuuu AAAAAA
DMACWHU<.f>	b,b,s12	00101bbb10110111FBBBssssss SSSSSS
DMACWHU<.cc><.f>	b,b,c	00101bbb11110111FBBBCCCCC 0QQQQQ
DMACWHU<.cc><.f>	b,b,u6	00101bbb11110111FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DMPYH

This section describes the standard DMPYH instruction. For the DSP version, see [DMPYH](#).

Function

Sum of dual 16x16 multiplication

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if(cc) {
    result = (b.h0 * c.h0) + (b.h1 * c.h1);
    a = result.w0;
    acc = result;
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. `w0` (lower) represents the 32-bit element of an operand. For more information about 64-bit operands, see “Data Formats”.

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

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
DMPYH<.f>	a,b,c	00101bbb00010000FBBBCCCCCAAAAAA
DMPYH<.f>	a,b,u6	00101bbb01010000FBBBuuuuuuAAAAAA
DMPYH<.f>	b,b,s12	00101bbb10010000FBBBssssssSSSSSS
DMPYH<.cc><.f>	b,b,c	00101bbb11010000FBBBCCCCC0QQQQQ
DMPYH<.cc><.f>	b,b,u6	00101bbb11010000FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DMPYHU

This section describes the standard DMPYHU instruction. For the DSP version, see [DMPYHU](#).

Function

Sum of dual unsigned 16x16 multiplication

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. w0 (lower) represents the 32-bit element of an operand. For more information about 64-bit operands, see [“Data Formats”](#).

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);
    a = result.w0 ;
    acc = result;
}
```

/* DMPYHU */

Assembly Code Example

```
DMPYHU r1,r2,r3    ; sum of dual 16x16 unsigned multiply of r2 and r3
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
DMPYHU<.f>	a,b,c	00101bbb00010001FBBBCCCCCAAAAAA
DMPYHU<.f>	a,b,u6	00101bbb01010001FBBBuuuuuuAAAAAA
DMPYHU<.f>	b,b,s12	00101bbb10010001FBBBsssssssSSSSSS
DMPYHU<.cc><.f>	b,b,c	00101bbb11010001FBBBCCCCC0QQQQQ
DMPYHU<.cc><.f>	b,b,u6	00101bbb11010001FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DMPYWH

Function

Sum of dual 32x16 multiplication. For the DSP version, see [DMPYWH](#).

Extension Group

MPY_OPTION == 9 or dsp_wide==true.

Operation

```
if (cc) {result = ((B.w0 * c.h0)+(B.w1 * c.h1)); A= result; acc = result;}
```



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 and B are 64-bit operands, and must be specified as even numbered registers. For more information about 64-bit operands, see [“Data Formats”](#).

Instruction Format

op A, B, c

Syntax Example

DMPYWH <.f> A,B,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	= Cleared

Description

Compute the sum of the two 32x16 products formed by multiplying B.w0 with c.h0, and B.w1 with c.h1. The resulting sum is assigned to the destination register pair, and is also assigned to the 64-bit accumulator (r59, r59).

In common with all operations that produce a 64-bit result, the A operand must be an even-numbered register (or the null destination).

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) {
    result = (B.w0 * c.h0) + (B.w1 * c.h1);
    A = result;
    acc = result;
}
```

/* DMPYWH */

Assembly Code Example

```
DMPYWH r0,r2,r5 ; sum of dual 32x16 multiply of r2 and r5
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
DMPYWH<.f>	a,b,c	00101bbb00110010FBBBCCCCCAAAAAA
DMPYWH<.f>	a,b,u6	00101bbb01110010FBBBuuuuuuAAAAAA
DMPYWH<.f>	b,b,s12	00101bbb10110010FBBBssssssSSSSSS
DMPYWH<.cc><.f>	b,b,c	00101bbb11110010FBBBCCCCC0QQQQQ
DMPYWH<.cc><.f>	b,b,u6	00101bbb11110010FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DMPYWHU

Function

Sum of dual unsigned 32x16 multiplication. For the DSP version, see [DMPYWHU](#).

Extension Group

MPY_OPTION == 9 or dsp_wide==true.

Operation

```
if (cc) {result = ((B.w0 * c.h0)+(B.w1 * c.h1)); A= result; acc = result;}
```



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 and B are 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see [“Data Formats”](#).

Instruction Format

op A,B,c

Syntax Example

DMPYWHU <.f> A,B,c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V	•	= Cleared

Description

Compute the sum of the two unsigned 32x16 products formed by multiplying B.w0 with c.h0, and B.w1 with c.h1. The resulting sum is assigned to the destination register pair, and is also assigned to the 64-bit accumulator (r59, r59).

In common with all operations that produce a 64-bit result, the A operand must be an even-numbered register (or the null destination).

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is always cleared.

Pseudo Code

```
if(cc) {                                     /* DMPYWHU */
    result =(B.w0 * c.h0)+(B.w1 * c.h1);
    A = result;
    acc = result;
}
```

Assembly Code Example

```
DMPYWHU r0,r2,r5    ; sum of dual unsigned 32x16 multiplication of (r3,r2)
                   ;and r5. The result is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
DMPYWHU<.f>	a,b,c	00101bbb00110011FBBBCCCCCAAAAA
DMPYWHU<.f>	a,b,u6	00101bbb01110011FBBBuuuuuuAAAAA
DMPYWHU<.f>	b,b,s12	00101bbb10110011FBBBsssssssSSSSSS
DMPYWHU<.cc><.f>	b,b,c	00101bbb11110011FBBBCCCCC0QQQQQ
DMPYWHU<.cc><.f>	b,b,u6	00101bbb11110011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

DSYNC

Function

Data transfer synchronization instruction

Extension Group

OS_OPT_OPTION == 1. This option cannot be configured in the ARC EM processor, and therefore this instruction is decoded as an illegal instruction in the ARC EM processor.

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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

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 therefore 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 epilogs.
- ❑ **Write:** store or push instructions including those from ENTER_S and interrupt or exception prologs.
- ❑ **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 u3          ; Data memory barrier
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
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

EI_S

Function

Execute Indexed

Extension Group

CODE_DENSITY

Operation

Execute instruction from table at a given index

Instruction Format

op u10

Syntax Example

EI_S u10

STATUS32 Flags Affected

Z	☐	= Dependent on instruction executed
N	☐	= Dependent on instruction executed
C	☐	= Dependent on instruction executed
V	☐	= Dependent on instruction executed

Description

The EI_S instruction can index up to 1,024 entries in the instruction table located at the EI_BASE register. Each entry in the table is typically a 32-bit encoded instruction. Because the u10 operand is a table index and each table entry is assumed to be 4 bytes, the u10 operand is scaled by a factor of 4 before being added to the table base address.

The processor jumps to the indexed table location, executes one instruction and then returns to the next instruction located sequentially after the EI_S.

Each instruction executed by the EI_S instruction from the table pointed to by EI_BASE is effectively in the execution slot of the EI_S, indicated by the STATUS32.ES bit. Instructions in the execution slot cannot be any of the following types: Bcc, BLcc, Jcc, JLcc, LPcc, RTIE, ENTER_S and LEAVE_S. All other instructions are permitted, and all operand types that are normally permitted for these instructions may be used by the instructions residing in the table. If long immediate data is used by an instruction in the table, the immediate data must immediately follow the instruction in the table.

An EI_S instruction raises an Illegal Instruction Sequence exception in the following cases:

- When JLI_S instruction is attempted in the execution slot of an EI_S instruction

- When any of the following instructions are attempted in the execution slot of an EI_S instruction:
 - Another EI_S instruction
 - Any jump or branch instruction (Bcc, BLcc, Jcc, JLcc)
 - A Loop instruction (LPcc)
 - A return from interrupt instruction (RTIE)
 - An ENTER_S or LEAVE_S instruction

**Note**

An illegal instruction sequence exception is not raised when the ENTER_S and LEAVE_S instructions have zero operands.

- When EI_S is the last instruction position of a zero-overhead loop
- When the EI_S instruction is in a branch or jump delay slot

Pseudo Code

```
BTA = PC + 2                      /* EI_S */
STATUS32.ES = 1
PC = EI_BASE + (u10 << 2)
```

Assembly Code Example

```
EI_S index    ; store the next PC in BTA, set the execution-slot status
               ;bit, and then jump to EI_BASE[index]
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
EI_S      u10    010111uuuuuuuuuu
```

ENTER_S

Function

Function Prolog Sequence

Extension Group

CODE_DENSITY

Operation

Push a collection of registers on to the stack, update stack pointer, and frame pointer, as required

Instruction Format

op u6

Syntax Example

ENTER_S {r13-r26, fp, blink}

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

This instruction executes the function prolog code, storing on the stack the set of registers defined by the 6-bit literal operand. On completion of this instruction, the stack pointer (r28) is decremented by 4S, where S is the total number of registers saved. If the frame-pointer is specified in the operand bit-vector, the stack-pointer is copied to the frame-pointer on completion of the operation.

For ENTER_S and LEAVE_S instructions, if the stack pointer is not 32-bit aligned, a misaligned access exception is always raised irrespective of the STATUS32.AD bit.

The status flags are not updated by this instruction.

The ENTER_S instruction is not allowed to appear in the delay slot of a jump or branch instruction. The processor raises an Illegal Instruction Sequence exception on any attempt to execute ENTER_S in a delay slot.



Note

An illegal instruction sequence exception is not raised when the ENTER_S instruction has zero operands.

The u6 encoding is as follows:

u[3:0]	Indicates the number of general-purpose registers to be saved on the stack, starting from r13 and counting contiguously upwards. If 32 general-purpose registers are configured, at most 14 registers (r13 to r26) can be saved. If only 16 general-purpose registers are configured, at most 3 registers (r13 to r15) can be saved. Therefore, if u[3:0] > 14 in a 32-register configuration, or if u[3:0] > 3 in a 16-register configuration, an Illegal Instruction exception is raised.
u[4]	If set to 1, r27 (frame-pointer) is saved, and on completion of the sequence, r27 is assigned the stack-pointer value (r28).
u[5]	If set to 1, the blink register (r31) is saved.

This instruction may perform as many as 16 writes to memory. These behave identically to the normal store instruction, such as:

```
ST r31, [sp, -64]
```

Pseudo Code

```

if (RGF_NUM_REGS == 32)                                /* ENTER_S */
    MaxRegs = 14
else
    MaxRegs = 3
if ((u[3:0] > MaxRegs)
    RaiseException (IllegalInstruction)

if (InDelaySlot)
    RaiseException (IllegalInstructionSequence)

S = u[3:0] + u[4] + u[5] // number of registers
W = 4
M = S * W           // stack space allocated

if (u[5] == 1)      // save blink?
{
    mem[sp - M] = r31
    M -= W
}
for (i = 0; i < u[3:0]; i++)
{
    mem[sp - M] = r[i+13] // save GPR i+13
    M -= W
}
if (u[4] == 1)      // save frame pointer?
{
    mem[sp - M] = r27
    M -= W
}

SP= sp - S*W        // allocate stack space
if (u[4] == 1)      // if frame-pointer saved
    r[27] = sp      // then set fp = sp

```

Assembly Code Example

```
ENTER_S {r13-r26, fp, blink} ;Push registers r13-r26, fp and blink to  
                               ;the stack, updating the stack pointer.  
                               ;Afterwards, the frame pointer assigns the  
                               ;new stack pointer value.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
ENTER_S    u6    110000UU111uuuu0
```

EX

Function

Atomic Exchange

Extension Group

BASELINE

Operation

$\text{temp} = b ; b = *c ; *c = \text{temp} ;$

Instruction Format

op b, c

Syntax Example

EX<.di> b,[c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

An atomic exchange operation, EX, is provided as a primitive for multiprocessor synchronization allowing the creation of semaphores in shared memory.

Two forms are provided: an uncached form (using the .DI directive) for synchronization between multiple processors in systems without cache-coherent shared memory, and a cached form for synchronization between processes on a single-processor system, or on a multiprocessor system with cache-coherent shared memory. The cached EX is completely handled by the core and does not depend on the AXI subsystem. The uncached EX relies on locked bus transactions that must be supported in the memory subsystem. Note that the locked bus transactions is supported in AXI3 but it is not supported in AXI4. For more information about this instruction, see the [Atomic Memory Operations](#) section.

The EX instruction exchanges the contents of the specified memory location with the contents of the specified register. This operation is atomic and the memory system ensures that the memory read and memory write cannot be separated by interrupts or by memory accesses from another source. The memory location must be 32-bit aligned (address bits [1:0] = 0). Thus, the EX-related accesses to data cache never cross a cache line boundary. Even if the data memory alignment checks are disabled

(STATUS32.AD bit is set to 1), an EX instruction to an unaligned memory address always raises an EV_Misaligned (Misaligned Data Access) exception.

The status flags are not updated with this instruction.

An immediate value is not permitted to be the destination of the exchange instruction. Using the long immediate indicator in the destination field, B=0x3E, raises a [Illegal Instruction](#) exception.



When used in conjunction with an MMU or MPU, both the read and write permissions must be set for EX to operate without causing a protection violation exception.

Pseudo Code

```
int32_t                                     /* EX */
swap( int32_t new_value, int32_t *word ) {
    int32_t  old_value;
    atomic {
        old_value = *word;
        *word     = new_value;
    }
    return( old_value );
}
```

Assembly Code Example

In this example, the processor attempts to get access to a shared resource by testing a semaphore against values 0 and 1.

- If the returned value is a 0, the resource is free and this device is now the owner.
- If the returned value is a 1, the resource is busy and the processor must wait till a 0 is returned.

The value 1 is always written to SEMAPHORE_ADDR. All processes trying to own the semaphore must write the same value.

The value at SEMAPHORE_ADDR must not be used to determine the current owner of the semaphore.

Example 9-1 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 9-2 To Release Semaphore using ST

```
release_resource:                ; Release Semaphore using ST
MOV R2, 0x00000000              ; indicates semaphore is free
ST R2, [SEMAPHORE_ADDR]         ; release semaphore
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
EX<.di> b, [c]	00100bbb00101111DBBBCCCCCC001100	
EX<.di> b, [u6]	00100bbb01101111DBBBuuuuuu001100	

EXTB

Function

Zero Extend Byte

Extension Group

BASELINE

Operation

$b = c \& 0x000000FF;$

Instruction Format

op b, c

Syntax Example

EXTB<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always Zero
C		= Unchanged
V		= Unchanged

Description

Zero extend the byte value in the source operand (c) and write the result into the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src & 0xFF          /* EXTB */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
EXTB r3,r0    ; Zero extend the bottom 8bits of r0 and write result to r3
```

Syntax and Encoding

For more information about the encodings, see “Key for 32-bit Addressing Modes and Encoding Conventions” and “Key for 16-bit Addressing Modes and Encoding Conventions” .

Instruction Code

EXTB<.f>	b,c	00100bbb00101111FBBBCCCCC000111
EXTB<.f>	b,u6	00100bbb01101111FBBBuuuuuu000111
EXTB_S	b,c	01111bbbccc01111

EXTH

Function

Zero Extend Half-word (16-bit)

Extension Group

BASELINE

Operation

$b = c \& 0x0000FFFF;$

Instruction Format

op b, c

Syntax Example

EXTH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always Zero
C		= Unchanged
V		= Unchanged

Description

Zero extend the 16-bit half-word value in the source operand (c) and write the result into the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src & 0xFFFF          /* EXTH */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
EXTH r3,r0    ; Zero extend the bottom 16 bits of r0 and write result to r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

EXTH<.f>	b,c	00100bbb00101111FBBBCCCCC001000
EXTH<.f>	b,u6	00100bbb01101111FBBBuuuuuu001000
EXTH_S	b,c	01111bbbccc10000

FCVT32

Function

Convert between two 32-bit data formats: single-precision float, signed or unsigned integer.

Alias

FS2INT, FS2UINT, FS2INT_RZ, FS2UINT_RZ, FINT2S, FUINT2S, FH2S, FS2H

Extension Group

FPU, FH2S and FS2H are available when `-fpu_hp_option==true`

Operation

```
a = (int) b;
a = (float) b;
a = (uint)b;
```

Instruction Format

op a, b, u6

Syntax Example

FCVT32 a,b,u6

STATUS32 Flags Affected

STATUS32 Flags

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	- Set on floating-point to integer conversion if: - Floating-point number is NaN - Floating-point number is infinity - Conversion result is outside the range of the integer data type (range is [0,65535] for unsigned short). - or - If any of the input operands is a signaling NaN when <code>FPU_CTRL.IVE==0</code> (invalid operation exceptions is not enabled).

DZ		= Unchanged
OF	•	Set when conversion from single precision to half precision results in a number too large to be represented (that is: the result is finite and has a magnitude that exceeds the maximum normalized number).
UF		= Unchanged
IX	•	Set when conversion from single precision to half precision results in a number too small to be represented (that is: the result after rounding has a magnitude less than the minimum normalized number), and it is inexact. or Set when conversion from single precision to half precision has a result that is rounded (that is: the result after rounding is different from the infinitely precise value before rounding).

Description

This instruction converts a 32-bit data format into another 32-bit data format. The type of conversion is determined by the `u6` operand. Depending on the value of `u6`, the processor does the following conversions: Bits 4 to 31 of the source 2 value passed to these instructions are ignored.

Instruction	u6 value	Description
FS2INT	FCVT32 a, b, 0b000011	Single-precision float to signed integer
FS2INT_RZ	FCVT32 a, b, 0b001011	Single-precision float to signed integer; round to zero
FINT2S	FCVT32 a, b, 0b000010	Signed integer to single-precision float
FS2UINT	FCVT32 a, b, 0b000001	Single-precision float to unsigned integer
FS2UINT_RZ	FCVT32 a, b, 0b001001	Single-precision float to unsigned integer; round to zero
FUINT2S	FCVT32 a, b, 0b000000	Unsigned integer to single-precision float
FH2S	FCVT32 a, b, 0b010101	Half-precision to single-precision conversion
FS2H	FCVT32 a, b, 0b010100	Single-precision to half-precision conversion

Pseudo Code

```
a = (float) b; /* FCVT32 */
```


Assembly Code Example

```
FCVT32 r2,r3, u6    ; conversion from r2 to r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FCVT32	a, b, c	00110bbb000010000BBBCCCCCAAAAAA
FCVT32	a, b, u6	00110bbb010010000BBBuuuuuuAAAAAA
FCVT32	b, b, s12	00110bbb100010000BBBssssssSSSSSS
FCVT32<.cc>	b, b, c	00110bbb110010000BBBCCCCC0QQQQQ
FCVT32<.cc>	b, b, u6	00110bbb110010000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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”](#).

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	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 signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	= 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 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FCVT32_64	a, b, c	00110 bbb 00 001001 0 BBB CCCCC AAAAAA
FCVT32_64	a, b, u6	00110 bbb 01 001001 0 BBB uuuuuu AAAAAA
FCVT32_64	b, b, s12	00110 bbb 10 001001 0 BBB ssssss SSSSSS
FCVT32_64<.cc>	b, b, c	00110 bbb 11 001001 0 BBB CCCCC 0QQQQQ
FCVT32_64<.cc>	b, b, u6	00110 bbb 11 001001 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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;

 **Note**

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”](#).

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	- 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. - Or - If any of the input operands to a floating-point instruction is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	= Unchanged
OF	= Unchanged
UF	= Unchanged
IX	Set when result is rounded in case of conversion from a long integer to a floating-point number

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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FCVT64	a, b, c	00110 bb 000 111 0000 BBB CCCCC AAAAA 0
FCVT64	a, b, u6	00110 bb 001 111 0000 BBB uuuuuu AAAAA 0
FCVT64	b, b, s12	00110 bb 010 111 0000 BBB ssssss SSSSSS
FCVT64<.cc>	b, b, c	00110 bb 011 111 0000 BBB CCCCC 0QQQQQ
FCVT64<.cc>	b, b, u6	00110 bb 011 111 0000 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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;
```



Note

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”](#)

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	- 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 signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	= Unchanged
OF	Set when double to single conversion results in a number too large to be represented as a float.
UF	Set when double to single conversion results in a number too small to be represented in single-precision format.
IX	Set when result is rounded

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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FCVT64_32	a, b, c	00110 bb 000 111 0010 BBB CCCCC AAAAA 0
FCVT64_32	a, b, u6	00110 bb 001 111 0010 BBB uuuuuu AAAAA 0
FCVT64_32	b, b, s12	00110 bb 010 111 0010 BBB ssssss SSSSS
FCVT64_32<.cc>	b, b, c	00110 bb 011 111 0010 BBB CCCCC 0QQQQQ
FCVT64_32<.cc>	b, b, u6	00110 bb 011 111 0010 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FDABS

Function

Double-precision floating-point absolute operation.

Alias

BCLR r0, r2, 0x1F

Extension Group

BASELINE

Operation

$B = (\text{double}) \text{ abs}(C);$

 **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”](#).

Instruction Format

op B, C

Syntax Example

FDABS 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                                /* 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 assembly 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

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”](#).

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	Set when the operands are in the following combinations: B= +infinity and C = -infinity or B= -infinity and C = +infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when 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 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](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
FDADD	a,b,c	00110 bb 000 1100010 BBB CCCCC AAAAA0
FDADD	a,b,u6	00110 bb 001 1100010 BBB uuuuuu AAAAA0
FDADD	b,b,s12	00110 bb 010 1100010 BBB sssssss SSSSSS

FDADD<.cc>	b, b, c	00110 bb 0111100010 BBB CCCCC 0 QQQQQ
FDADD<.cc>	b, b, u6	00110 bb 0111100010 BBB uuuuuu 1 QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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”](#).

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 not 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](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
FDCMP	b, c	00110 bb 0001100111BBBCCCCC000000
FDCMP	b, u6	00110 bb 0011100111BBBuuuuuu000000
FDCMP	b, s12	00110 bb 0101100111BBBssssssSSSSSS

FDCMP<.cc>	b, c	00110 bb 011 110011 1 BBB CCCCC 0 QQQQQ
FDCMP<.cc>	b, u6	00110 bb 011 110011 1 BBB uuuuuu 1 QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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 signaling 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”](#) .

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 not 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.



Note

If the source operands have a magnitude of zero (0), the Z flag is always updated irrespective of the sign of the source operands.

Pseudo Code

```
if(cc) {
    Z_flag = (src1 == src2) && (src1 != NaN) && (src2 != NaN);
    N_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    C_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    V_flag = (src1 == NaN ) || (src2 == NaN);
    FPU_STATUS.IV = V_flag;
}
```

/* FDCMPF */

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](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code

FDCMPF	b, c	00110 bb 0001101001BBBCCCCC000000
FDCMPF	b, u6	00110 bb 0011101001BBBuuuuuu000000
FDCMPF	b, s12	00110 bb 0101101001BBBssssssSSSSSS

FDCMPF<.cc>	b, c	00110 bb 011 110100 1 BBB CCCCC 0 QQQQQ
FDCMPF<.cc>	b, u6	00110 bb 011 110100 1 BBB uuuuuu 1 QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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”](#).

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	Set when the operands are in the following combinations: 0/0 or infinity/infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Set when the divisor is zero

OF	• Set when the normalized result exceeds the maximum representable number in a 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FDDIV	a,b,c	00110 bb 000 110111 0 BBB CCCCC AAAAA 0
FDDIV	a,b,u6	00110 bb 001 110111 0 BBB uuuuuu AAAAA 0
FDDIV	b,b,s12	00110 bb 010 110001 0 BBB ssssss SSSSSS
FDDIV<.cc>	b,b,c	00110 bb 011 110111 0 BBB CCCCC 0QQQQQ
FDDIV<.cc>	b,b,u6	00110 bb 011 110111 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#)

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));
}
```



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”](#).

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	- Set when the B and C operands are in the following combinations: $0 * \text{infinity}$ or $\text{infinity} * 0$ - Set when the A, B, and C operands are in the following combination: $(B * C) = +\text{infinity}$ and $A = -\text{infinity}$ or $(B * C) = -\text{infinity}$ and $A = +\text{infinity}$ - If any of the input operands is a signaling NaN when <code>FPU_CTRL.IVE==0</code> (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a 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 after the accumulation

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.



Note

This instruction uses the accumulator as described in “[Extension Core Registers](#)”.

Pseudo Code

```
if(cc) {
    A = (double) (ACC + (B * C));
}
```

/* FDMADD */

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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FDMADD	a, b, c	00110 bb 000 110101 0 BBB CCCCC AAAAA 0
FDMADD	a, b, u6	00110 bb 001 110101 0 BBB uuuuuu AAAAA 0
FDMADD	b, b, s12	00110 bb 010 110101 0 BBB ssssss SSSSSS
FDMADD<.cc>	b, b, c	00110 bb 011 110101 0 BBB CCCCC 0QQQQQ
FDMADD<.cc>	b, b, u6	00110 bb 011 110101 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#)

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));
}
```



Note

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”](#).

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	- Set when the B and C operands are in the following combinations: $0 * \text{infinity}$ or $\text{infinity} * 0$ - Set when the A, B, and C operands are in the following combination: $(B * C) = +\text{infinity}$ and $A = +\text{infinity}$ or $(B * C) = -\text{infinity}$ and $A = -\text{infinity}$ - If any of the input operands is a signaling NaN when <code>FPU_CTRL.IVE==0</code> (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a double-precisions 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 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.



Note

This instruction uses the accumulator as described in “[Extension Core Registers](#)”.

Pseudo Code

```
if(cc) {
    A = (double) (ACC - (B * C));
}
```

/* FDMSUB */

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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FDMSUB	a, b, c	00110 bb 000 110110 0 BBB CCCCC AAAAA 0
FDMSUB	a, b, u6	00110 bb 001 110110 0 BBB uuuuuu AAAAA 0
FDMSUB	b, b, s12	00110 bb 010 110110 0 BBB ssssss SSSSS S
FDMSUB<.cc>	b, b, c	00110 bb 011 110110 0 BBB CCCCC 0QQQQQ Q
FDMSUB<.cc>	b, b, u6	00110 bb 011 110110 0 BBB uuuuuu 1QQQQQ Q

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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”](#).

Instruction Format

op A,B,C

Syntax Example

```
FDMUL A,B,C  
FDMUL<.cc>A,B,C
```

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	- Set when (0*infinity) or (infinity*0) operations occur - Or - If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	= Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
FDMUL	a,b,c	00110 bb 000 1100000 0 BBB CCCCC AAAAA 0
FDMUL	a,b,u6	00110 bb 001 1100000 0 BBB uuuuuu AAAAA 0
FDMUL	b,b,s12	00110 bb 010 1100000 0 BBB ssssss SSSSSS
FDMUL<.cc>	b,b,c	00110 bb 011 1100000 0 BBB CCCCC 0QQQQQ
FDMUL<.cc>	b,b,u6	00110 bb 011 1100000 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FDNEG

Function

Double-precision floating-point negate operation.

Alias

BXOR r0, r2, 0x1F

Extension Group

BASELINE

Operation

$B = (\text{double}) \text{ neg}(C) ;$



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”](#).

Instruction Format

op B, C

Syntax Example

FDNEG 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                                /* 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 assembly 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

FDSQRT

Function

Double-precision floating-point square-root

Extension Group

(HAS_FPU == 1) && (FPU_DP_OPTION == 1) && (FPU_DIV_OPTION == 1)

Operation

$B = (\text{double}) \text{ sqrt}(C);$



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”](#).

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” and “Key for 16-bit Addressing Modes and Encoding Conventions” .

Instruction Code

FDSQRT	b, c	00110 bb 000 101111 0 BBB CCCCC 000001
FDSQRT	b, u6	00110 bb 001 101111 0 BBB uuuuuu 000001

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”](#).

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	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a 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 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](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
FDSUB	a,b,c	00110 bb 000 110010 0 BBB CCCCC AAAAA 0
FDSUB	a,b,u6	00110 bb 001 110010 0 BBB uuuuuu AAAAA 0
FDSUB	b,b,s12	00110 bb 010 110010 0 BBB ssssss SSSSSS
FDSUB<.cc>	b,b,c	00110 bb 011 110010 0 BBB CCCCC 0QQQQQ
FDSUB<.cc>	b,b,u6	00110 bb 011 110010 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FFS

Function

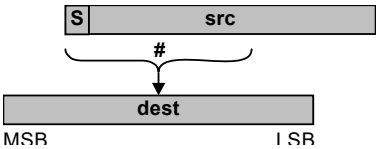
Find First Set

Extension Group

BITSCAN_OPTION

Operation

dest ← bit position of first 1 in source operand, from bit 0 upwards



Instruction Format

op b,c

Syntax Example

FFS<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Returns the bit position of the least-significant (that is, lowest numbered) non-zero bit in the source operand. Examples of returned values are shown in the following table. The '-' symbol signifies any hexadecimal digit:

Operand Value	Result	Z	N
0x-----1	0	0	0
0x-----2	1	0	0
0x-----4	2	0	0

Operand Value	Result	Z	N
0x-----8	3	0	0
:	:	:	:
0x40000000	30	0	0
0x80000000	31	0	1
0x00000000	31	1	0

Pseudo Code

```

s = 0                                /* FFS */

while (src && (((src >> s) & 1) == 0))
    ++s;

dest = (src == 0) ? 31 : s;

if (F == 1) then {
    Z_flag = (src == 0) ? 1 : 0;
    N_flag = src[31];
}

```

Assembly Code Example

```

FFS r1,r2    ; Find least-significant non-zero bit in r2 write result
              into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```

FFS<.f>    b,c    00101bbb00101111FBBBCCCCC010010
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)

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 the H flag is set (halt processor flag), all other flag states are ignored and are not updated.

The arithmetic flags in the STATUS32 register (auxiliary address 0x0A) are always updated by the FLAG instruction. The flag setting field, F, is always encoded as 0 for this instruction.

Only the Z, N, C and V flags are modified by a FLAG instruction in user mode; all other flags remain unchanged.

The FLAG instruction is serializing – ensuring that no further instructions can be completed before all flag updates take effect.

Bits U, DE, AE, RB, IE, and L in the STATUS32 register are not modified by the FLAG instruction in any operating mode. These are updated when the processor changes state because of branches, interrupts, or exceptions, and when the processor returns from interrupts or exception by executing the RTIE instruction.

Pseudo Code

```

FlushPipe()                                     /* FLAG */
if ((src[0] == 1) && (STATUS32[7] == 0)) then
    if (HAS_INTERRUPTS > 0)
        STATUS32[0] = 1
        Halt()
    else
        STATUS32[11:8] = src[11:8]
        if (STATUS32[7] == 0 && HAS_INTERRUPTS > 0) /*in kernel mode*/
            STATUS32[4:1] = src[4:1]
            if (DIV_REM_OPTION == 1)
                STATUS32[13] = src[13]
            if (STACK_CHECKING == 1)
                STATUS32[14] = src[14]
            if (LL64_OPTION == 1)
                STATUS32[19] = src[19]
            STATUS32[20] = src[20]

```

Assembly Code Example

```
FLAG 1                ; Halt processor (other flags
                       ; not updated)
FLAG 6                ; Set the Interrupt threshold to
                       ; 3. Any interrupt with equal or
                       ; higher priority than 3 is
                       ; serviced if interrupts are
                       ; enabled.
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
FLAG	c	00100rrr001010010RRRCCCCCRRRRRR
FLAG	u6	00100rrr011010010RRRuuuuuuRRRRRR
FLAG	s12	00100rrr101010010RRRssssssSSSSSS
FLAG<.cc>	c	00100rrr111010010RRRCCCCC0QQQQQ
FLAG<.cc>	u6	00100rrr111010010RRRuuuuuu1QQQQQ

FLS

Function

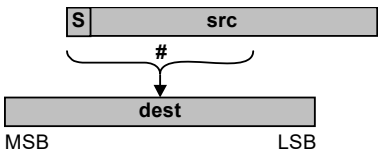
Find Last Set

Extension Group

BITSCAN_OPTION

Operation

dest ← bit position of last 1 in source operand, from bit 0 upwards



Instruction Format

op b,c

Syntax Example

FLS<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Returns the bit position of the most-significant non-zero bit in the source operand. If the source operand is 0 (zero), the value 0 (zero) is returned. Examples of returned values are shown in the following table. The ‘-’ symbol signifies any hexadecimal digit:

Operand Value	Result	Z	N
Any value greater than 0x80000000 (inclusive)	31	0	1
Any value between 0x40000000 (inclusive) and 0x80000000	30	0	0

Operand Value	Result	Z	N
Any value between 0x20000000 (inclusive) and 0x40000000	29	0	0
:	:	:	:
Any value between 0x00000004 (inclusive) and 0x00000008	2	0	0
Any value between 0x00000002 (inclusive) and 0x00000004	1	0	0
0x00000001	0	0	0
0x00000000	0	1	0

Pseudo Code

```

s = 31                                /* FLS */

while (src && (((src >> s) & 1) == 0))
    --s;

dest = s;

if (F == 1) then {
    Z_flag = (src == 0) ? 1 : 0;
    N_flag = src[31];
}

```

Assembly Code Example

```

FLS r1,r2    ; Find most-significant non-zero bit in r2 and write
              ;result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```

FLS<.f>    b,c    00101bbb00101111FBBBCCCCC010011
FLS<.f>    b,u6   00101bbb01101111FBBBuuuuuu010011

```

FSABS

Function

Single-precision floating-point absolute operation.

Alias

BCLR r1, r3, 0x1F

Extension Group

BASELINE

Operation

b = (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 assembly alias. For syntax and encoding information, refer to the BCLR instruction used in the following format:

```
BCLR r1, r2, 0x1F
```

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

FSADD

Function

Single-precision floating-point addition

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {
    a = (float) (b + c);
}
```

Instruction Format

op a,b,c

Syntax Example

FSADD a,b,c

FSADD<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	Set when the operands are in the following combinations: b= +infinity and c = -infinity or b= -infinity and c = +infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The sum of b and c operands is stored in the destination register.

Pseudo Code

```
if(cc) {                                     /* FSADD */
    a = (float) (b + c);
}
```

Assembly Code Example

```
FSADD r1,r2,r3    ; floating-point addition of r2 and r3 and the
                  ; result is stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
FSADD	a,b,c	00110bbb00000010BBBCCCCCAAAAAA
FSADD	a,b,u6	00110bbb010000010BBBuuuuuuAAAAAA
FSADD	b,b,s12	00110bbb100000010BBBsssssssSSSSSS
FSADD<.cc>	b,b,c	00110bbb110000010BBBCCCCC0QQQQQ

FSADD<.cc> b,b,u6 00110bbb110000010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSCMP

Function

Single-precision floating-point comparison

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {
    compare(b , c);
}
```

Instruction Format

op b,c

Syntax Example

```
FSCMP b,c
FSCMP<.cc> b,c
```

STATUS32 Flags Affected

Z	•	= Set when b and c are equal
N	•	= Set when b is less than c
C	•	= Set when b is less than c
V	•	Set when the comparison result is unordered. If this bit is set, the Z, N, C flags are cleared.

FPU_STATUS Flags

IV	•	Set when either of the inputs is a signaling NaN and FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The b and c operands are compared and the core flags are updated.

 **Note**

If the source operands have a magnitude of zero (0), the Z flag is always updated irrespective of the sign of the source operands.

Pseudo Code

```
if(cc) { /* FSCMP */
    Z_flag = (src1 == src2) && (src1 != NaN) && (src2 != NaN);
    N_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    C_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    V_flag = (src1 == NaN) || (src2 == NaN);
    FPU_STATUS.IV = (src1 == sNaN) || (src2 == sNaN);
}
```

Assembly Code Example

```
FSCMP r2,r3      ; floating-point comparison of r2 and r3 and the
                  ; core flags are updated
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
FSCMP	b, c	00110bbb00000111BBBCCCCC000000
FSCMP	b, u6	00110bbb010000111BBBuuuuuu000000
FSCMP	b, s12	00110bbb100000111BBBssssssSSSSSS
FSCMP<.cc>	b, c	00110bbb110000111BBBCCCCC0QQQQQ
FSCMP<.cc>	b, u6	00110bbb110000111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSCMPF

Function

Single-precision floating-point comparison operation with IEEE 754 flag generation. This instruction is similar to the FSCMP instruction in cases when either of the instruction operands is a signaling NaN. The FSCMPF instruction updates the invalid flag (FPU_STATUS.IV) when either of the operands is a quiet or signaling NaN, whereas, the FSCMP instruction updates the invalid flag (FPU_STATUS.IV) only when either of the operands is a signaling NaN.

Extension Group

FPU

Operation

```
if (cc) {
    compare(b , c);
}
```

Instruction Format

op b,c

Syntax Example

```
FSCMPF b,c
FSCMPF<.cc> b,c
```

STATUS32 Flags Affected

Z	•	= Set when b and c are equal
N	•	= Set when b is less than c
C	•	= Set when b is less than c
V	•	= Set when the comparison result is unordered. If this bit is set, the Z, N, C flags are cleared

FPU_STATUS Flags

IV	•	Set when either of the inputs is a quiet or signaling NaN and FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The b and c operands are compared and the core flags are updated.



If the source operands have a magnitude of zero (0), the Z flag is always updated irrespective of the sign of the source operands.

Pseudo Code

```
if(cc) {
    Z_flag = (src1 == src2) && (src1 != NaN) && (src2 != NaN);
    N_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    C_flag = (src1 < src2) && (src1 != NaN) && (src2 != NaN);
    V_flag = (src1 == NaN ) || (src2 == NaN);
    FPU_STATUS.IVF = V_flag;
}
```

Assembly Code Example

```
FSCMPF r2,r3      ; floating-point comparison of r2 and r3 and update flags
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
FSCMPF	b, c	00110bbb000001001BBBCCCCC000000
FSCMPF	b, u6	00110bbb010001001BBBuuuuuu000000
FSCMPF	b, s12	00110bbb100001001BBBssssssSSSSSS

FSCMPF<.cc>	b, c	00110bbb110001001BBBCCCCC0QQQQQ
FSCMPF<.cc>	b, u6	00110bbb110001001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSDIV

Function

Single-precision floating-point division

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {
    a = (float) (b / c);
}
```

Instruction Format

op a,b,c

Syntax Example

FSDIV a,b,c

FSDIV<cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	Set when the operands are in the following combinations: 0/0 or infinity/infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Set when the divisor is zero
OF	Set when the normalized result exceeds the maximumrepresentable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The b operand is divided by the c operand, and the result is stored in the destination register.

Pseudo Code

```
if(cc) {                                     /* FSDIV */
    a = (float) (b / c);
}
```

Assembly Code Example

```
FSDIV r1,r2,r3    ; floating-point division of r2 by r3 and the
                  ; result is stored in r1.
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
FSDIV	a,b,c	00110bbb000001110BBBCCCCCAAAAAA
FSDIV	a,b,u6	00110bbb010001110BBBuuuuuuAAAAAA
FSDIV	b,b,s12	00110bbb100000010BBBsssssssSSSSSS
FSDIV<.cc>	b,b,c	00110bbb110001110BBBCCCCC0QQQQQ

FSDIV<.cc> b,b,u6 00110bbb110001110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSMADD

Function

Single-precision floating-point fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply-add operation performed in one step, with a single rounding.

Extension Group

(HAS_FPU == 1) && (FPU_FMA_OPTION == 1)

Operation

```
if (cc) {
    a = (float) (acc + (b * c));
}
```

Instruction Format

op a,b,c

Syntax Example

FSMADD a,b,c

FSMADD<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	- Set when the b and c operands are in the following combinations: $0 * \text{infinity}$ or $\text{infinity} * 0$ - Set when the a, b, and c operands are in the following combinations: $(b * c) = +\text{infinity}$ and $a = -\text{infinity}$ or $(b * c) = -\text{infinity}$ and $a = +\text{infinity}$ - If any of the input operands is a signaling NaN when <code>FPU_CTRL.IVE==0</code> (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded after the accumulation

Description

The product of b and c operands is computed, added to the accumulator, and the resulting sum is rounded and stored in the destination register.



Note

This instruction uses the accumulator as described in “[Extension Core Registers](#)”.

Pseudo Code

```
if(cc) {
    a = (float) (acc + (b * c));
}
```

/* FSMADD */

Assembly Code Example

```
FSMADD r1,r2,r3    ; floating-point multiplication of r2 and r3 and the
                   ; product is accumulated and the sum stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FSMADD	a, b, c	00110 bbb 00 000101 0 BBB CCCCC AAAAAA
FSMADD	a, b, u6	00110 bbb 01 000101 0 BBB uuuuuu AAAAAA
FSMADD	b, b, s12	00110 bbb 10 000101 0 BBB ssssss SSSSSS
FSMADD<.cc>	b, b, c	00110 bbb 11 000101 0 BBB CCCCC 0QQQQQ
FSMADD<.cc>	b, b, u6	00110 bbb 11 000101 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSMSUB

Function

Single-precision floating-point fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply-subtract operation performed in one step, with a single rounding.

Extension Group

(HAS_FPU == 1) && (FPU_FMA_OPTION == 1)

Operation

```
if (cc) {
    a = (float) (acc - (b * c));
}
```

Instruction Format

op a,b,c

Syntax Example

```
FSMSUB a,b,c
FSMSUB<.cc> a,b,c
```

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	- Set when the b and c operands are in the following combinations: $0 * \text{infinity}$ or $\text{infinity} * 0$ - Set when the a, b, and c operands are in the following combinations: $(b * c) = +\text{infinity}$ and $a = +\text{infinity}$ or $(b * c) = -\text{infinity}$ and $a = -\text{infinity}$ - If any of the input operands is a signaling NaN when <code>FPU_CTRL.IVE==0</code> (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The product of b and c operands is computed, subtracted from the accumulator; the result is rounded and stored in the destination register.



Note

This instruction uses the accumulator as described in “[Extension Core Registers](#)”.

Pseudo Code

```
if(cc) {
    a = (float) (acc - (b * c));
}
```

/* FSMSUB */

Assembly Code Example

```
FSMSUB r1,r2,r3    ; floating-point multiplication of r2 and r3 and the
                   ; product is subtracted from the accumulator and result is
                   ; stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
FSMSUB	a, b, c	00110bbb000001100BBBCCCCCAAAAAA
FSMSUB	a, b, u6	00110bbb010001100BBBuuuuuuAAAAAA
FSMSUB	b, b, s12	00110bbb100001100BBBssssssSSSSSS
FSMSUB<.cc>	b, b, c	00110bbb110001100BBBCCCCC0QQQQQ
FSMSUB<.cc>	b, b, u6	00110bbb110001100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSMUL

Function

Single-precision floating-point multiply

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {
    a = (float) (b * c);
}
```

Instruction Format

op a,b,c

Syntax Example

FSMUL a,b,c

FSMUL<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	- Set when (0*infinity) or (infinity*0) operations occur - Or - If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	= Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The b and c operands are multiplied and the product is stored in the destination register.

Pseudo Code

```
if(cc) {
    a = (float) (b * c);
}
```

/* FSMUL */

Assembly Code Example

```
FSMUL r1,r2,r3    ; floating-point multiplication of r2 and r3 and the
                  ; result is stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

FSMUL<.f>	a, b, c	00110bbb00000000BBBCCCCCAAAAAA
FSMUL<.f>	a, b, u6	00110bbb010000000BBBuuuuuuAAAAAA
FSMUL<.f>	b, b, s12	00110bbb100000000BBBsssssssSSSSSS
FSMUL<.cc>	b, b, c	00110bbb110000000BBBCCCCC0QQQQQ
FSMUL<.cc>	b, b, u6	00110bbb110000000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

FSNEG

Function

Single-precision floating-point negate operation.

Alias

BXOR r1, r2, 0x1F

Extension Group

BASELINE

Operation

b = (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 assembly alias. For syntax and encoding information, refer to the BXOR instruction used in the following format:

```
BXOR r1, r2, 0x1F
```

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

FSSQRT

Function

Single-precision floating-point square-root

Extension Group

(HAS_FPU == 1) && (FPU_DIV_OPTION == 1)

Operation

```
if (cc) {  
    b = (float) sqrt(c);  
}
```

Instruction Format

op b,c

Syntax Example

FSSQRT b,c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	Set when the c operand is less then -zero Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	= Unchanged
OF	= Unchanged
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The b operand contains the square-root of the c operand.

Pseudo Code

```
b = (float) sqrt(c);          /* FSSQRT */
```

Assembly Code Example

```
FSSQRT r2,r3      ; floating-point square-root of r3 is stored in r2
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
FSSQRT	b, c	00110bbb001011110BBBCCCCC000000
FSSQRT	b, u6	00110bbb011011110BBBuuuuuu000000

FSSUB

Function

Single-precision floating-point subtraction

Extension Group

HAS_FPU == 1

Operation

```
if (cc) {  
    a = (float) (b - c);  
}
```

Instruction Format

op a,b,c

Syntax Example

FSSUB a,b,c

FSSUB<.cc> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

FPU_STATUS Flags

IV	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signaling NaN when FPU_CTRL.IVE==0 (invalid operation exceptions is not enabled).
DZ	Unchanged
OF	Set when the normalized result exceeds the maximum representable number in a single-precision format
UF	Set when the result is too small to be represented in a single-precision format
IX	Set when result is rounded

Description

The difference of b and c operands is stored in the destination register.

Pseudo Code

```
if(cc) {                                     /* FSSUB */
    a = (float) (b - c);
}
```

Assembly Code Example

```
FSSUB r1,r2,r3    ; floating-point subtraction of r3 from r2; and the
                  ; result is stored in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
FSSUB	a,b,c	00110bbb000000100BBBCCCCCAAAAAA
FSSUB	a,b,u6	00110bbb010000100BBBuuuuuuAAAAAA
FSSUB	b,b,s12	00110bbb100000100BBBsssssssSSSSSS
FSSUB<.cc>	b,b,c	00110bbb110000100BBBCCCCC0QQQQQ
FSSUB<.cc>	b,b,u6	00110bbb110000100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

J

Function

Jump unconditionally

Extension Group

BASELINE

Operation

PC = c;

Instruction Format

op c

Syntax Example

J [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Use this instruction to jump to the new program counter address that is specified as the absolute address in the source operand (c). Jump instructions can target any address within the full memory address map, but the target address is always 16-bit aligned. Because the delay slot mode controls the execution of the instruction which is in the delay slot, the instruction must never be the target of any branch or jump instruction.



Caution

The J and J_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

Returning from an branch-and-link or jump-and-link is accomplished by jumping to the contents of the BLINK register (see [Link Registers, ILINK \(r29\), BLINK \(r31\)](#)), using the J [BLINK] instruction.

The particular delay slot modes for the jump instructions are:

Delay Slot Mode	Description
J	Never execute next instruction
J_S	Never execute next instruction
J.D	Always execute next instruction
J_S.D	Always execute next instruction

[Table 6-37](#) on page [291](#) provides more information about the delay slot modes, .d.
[Table 6-7](#) on page [264](#) provides information about the condition codes, cc.

Description

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Indexed jump or execute instructions (JLI_s and EI_S)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```
if N==1 then                                /* J */
    DelaySlot(nPC)
PC = src
```

Assembly Code Example

```
J [r1]                                     ; jump to address in r1
NOP
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
J	[c]	00100RRR00100000RRRRCCCCCRRRRRR
J	u6	00100RRR01100000RRRRuuuuuuRRRRRR

J	s12	00100RRR10100000RRRRssssssSSSSSS
J.D	[c]	00100RRR00100001RRRRCCCCCRRRRRR
J.D	u6	00100RRR01100001RRRRuuuuuuRRRRRR
J.D	s12	00100RRR10100001RRRRssssssSSSSSS
J_S	[b]	01111bbb00000000
J_S.D	[b]	01111bbb00100000
J_S	[BLINK]	0111111011100000
J_S.D	[BLINK]	0111111111100000

Jcc

Function

Jump Conditionally

Extension Group

BASELINE

Operation

if (cc) PC = c;

Instruction Format

op c

Syntax Example

Jcc [c]

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

If the specified condition is met (cc = true), the program execution is resumed from the new program counter address that is specified as the absolute address in the source operand (c). Jump instructions can target any address within the full memory address map, but the target address is always 16-bit aligned. Because the delay slot mode controls the execution of the instruction which is in the delay slot, the instruction must never be the target of any branch or jump instruction.



Caution

The Jcc and J_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The Jcc.D instruction may not use a long-immediate value as the target address operand.

Returning from an branch-and-link or jump-and-link is accomplished by jumping to the contents of the BLINK register (see [Link Registers, ILINK \(r29\), BLINK \(r31\)](#)), using the Jcc [BLINK] instruction.

The particular delay slot modes for the jump instructions are:

Delay Slot Mode	Description
JEQ_S	Only execute next instruction when <i>not</i> branching
JNE_S	Only execute next instruction when <i>not</i> branching
Jcc.D	Always execute next instruction

[Table 6-37](#) on page [291](#) provides more information about the delay slot modes, .d.

[Table 6-7](#) on page [264](#) provides information about the condition codes, cc.

Description

The processor raises an [Illegal Instruction Sequence](#) exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt ([RTIE](#))
- Any instruction with long-immediate data as a source operand

Pseudo Code

```
if cc==true then                                /* Jcc */
  if N==1 then
    DelaySlot(nPC)
    PC = src
  else
    PC = nPC
```

Assembly Code Example

```
JEQ [r1]    ; jump to address in r1 if the Z flag is set
NOP
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
Jcc	[c]	00100RRR11100000RRRCCCCC0QQQQQ
Jcc	u6	00100RRR11100000RRRuuuuuu1QQQQQ

Jcc.D	[c]	00100RRR11100001RRRRCCCCC0QQQQQ
Jcc.D	u6	00100RRR11100001RRRRuuuuuu1QQQQQ
JEQ_S	[BLINK]	0111110011100000
JNE_S	[BLINK]	0111110111100000

JL

Function

Jump and Link unconditionally

Extension Group

BASELINE

Operation

PC = c; BLINK = NEXT_PC;

Instruction Format

op c

Syntax Example

JL [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

When this instruction is executed, the program execution is resumed from the new program counter address that is specified as the absolute address in the source operand (c). Jump and link instructions can have any target address within the full memory address map, but the target address is 16-bit aligned. In parallel, the program counter address (PC) that immediately follows the jump instruction is written into the BLINK register (R31). Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction.



Caution

The JL and JL_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The JL.D instruction may not use a long-immediate value as the target address operand.

The particular delay slot modes for the jump and link instructions are:

Delay Slot Mode	Description
JL	Never execute next instruction
JL_S	Never execute next instruction
JL.D	Always execute next instruction
JL_S.D	Always execute next instruction

[Table 6-37](#) on page [291](#) provides more information about the delay slot modes, .d.

[Table 6-7](#) on page [264](#) provides more information about the condition codes, cc.

An [Illegal Instruction Sequence](#) exception is raised if an executed delay slot contains any of the following:

- ❑ Another jump or branch instruction
- ❑ Conditional loop instruction (LPcc)
- ❑ Indexed jump or execute instructions (JLI_s and EI_S)
- ❑ Return from interrupt ([RTIE](#))
- ❑ Any instruction with long-immediate data as a source operand

Pseudo Code

```
if N==1 then                                /* JL */
    BLINK = dPC
    DelaySlot(nPC)
else
    BLINK = nPC
PC = src;
```

Assembly Code Example

```
JL [r1]      ; jump and link to address
              ; in r1 and store the return address in BLINK
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
JL	[c]	00100RRR00100010RRRCCCCCRRRRRR
JL	u6	00100RRR01100010RRRuuuuuuRRRRRR

JL	s12	00100RRR10100010RRRRssssssSSSSSS
JL.D	[c]	00100RRR00100011RRRRCCCCCRRRRRR
JL.D	u6	00100RRR01100011RRRRuuuuuuRRRRRR
JL.D	s12	00100RRR10100011RRRRssssssSSSSSS
JL_S	[b]	01111bbb01000000
JL_S.D	[b]	01111bbb01100000

JLcc

Function

Jump and Link Conditionally

Extension Group

BASELINE

Operation

if (cc) {PC = c; BLINK = NEXT_PC;}

Instruction Format

op c

Syntax Example

JLcc [c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

If the specified condition is met (cc = true), the program execution is resumed from the new program counter address that is specified as the absolute address in the source operand (c). Jump and link instructions can have any target address within the full memory address map, but the target address is 16-bit aligned. In parallel, the program counter address (PC) that immediately follows the jump instruction is written into the BLINK register (R31). Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction.



Caution

The JLcc and JL_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction. Do not place branch or jump instructions in the execution slot of EI_S instructions.

The JLcc.D instruction may not use a long-immediate value as the target address operand

The particular delay slot modes for the jump and link instructions are:

Table 9-2 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 6-37](#) on page 291 provides more information about the delay slot modes, .d.

[Table 6-7](#) on page 264 provides more information about the condition codes, cc.

An [Illegal Instruction Sequence](#) exception is raised if an executed delay slot contains any of the following:

- ❑ Another jump or branch instruction
- ❑ Conditional loop instruction (LPcc)
- ❑ Return from interrupt ([RTIE](#))
- ❑ Any instruction with long-immediate data as a source operand

Pseudo Code

```

if cc==true then                                /* JLcc */
  if N==1 then
    BLINK = dPC
    DelaySlot(nPC)
  else
    BLINK = nPC
  PC = src
else
  PC = nPC

```

Assembly Code Example

```

JLEQ [r1]    ; if the Z flag is set then jump and link to address
              ; in r1 and store the return address in BLINK

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

JLcc	[c]	00100RRR11100010RRRRCCCCC0QQQQQ
JLcc	u6	00100RRR11100010RRRRuuuuuu1QQQQQ
JLcc.D	[c]	00100RRR11100011RRRRCCCCC0QQQQQ
JLcc.D	u6	00100RRR11100011RRRRuuuuuu1QQQQQ

JLI_S

Function

Jump and Link Indexed

Extension Group

CODE_DENSITY

Operation

$BLINK = next\_PC; PC = JLI\_BASE + (u10 \ll 2);$

Instruction Format

op u10

Syntax Example

JLI_S u10

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The address of the next instruction is stored in the link register BLINK (R31). Program execution then resumes from the location given by the contents of the JLI_BASE auxiliary register plus the unsigned 10-bit index operand shifted left by two bit positions.

The JLI_S instruction can index up to 1,024 entries in the jump table defined by the JLI_BASE register. Each entry in the table is typically a 32-bit encoded relative unconditional branch instruction. Because the u10 operand is a table index and each table entry is assumed to be 4 bytes, the u10 operand is scaled by a factor of 4 before being added to the table base address.

Pseudo Code

```
BLINK = PC + 2                                /* JLI_S */
PC = JLI_BASE + (u10 << 2)
```

Assembly Code Example

```
JLI_S index      ; store the return address in BLINK, and  
                  ; then jump to JLI_BASE[index]
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
JLI_S u10 010110uuuuuuuuuu
```

KFLAG

Function

Sets kernel Flags

Extension Group

BASELINE

Operation

if (cc) assign bits from operand to STATUS32

Instruction Format

op c

Syntax Example

KFLAG<.cc> c

STATUS32 Flags Affected

Flag		Mode	Source of operand
IE	•	Kernel	Bit 31 of Source Operand
US	•	Kernel	Bit 20 of Source Operand
AD	•	Kernel	Bit 19 of Source Operand
RB[2:0]	•	Kernel	Bits [18:16] of Source Operand, if RGF_NUM_BANKS > 1. If RGF_NUM_BANKS == 1, RB[2:0] is read as zero and ignored on write.
SC	•	Kernel	Bit 14 of Source Operand if stack checking is configured
DZ	•	Kernel	Bit 13 of Source Operand, if DIV_REM_OPTION is configured
L			Unchanged
Z	•	Any	Bit 11 of Source Operand
N	•	Any	Bit 10 of Source Operand
C	•	Any	Bit 9 of Source Operand
V	•	Any	Bit 8 of Source Operand
U			Unchanged
DE			Unchanged

AE	•	Kernel	Bit 5 of Source Operand
E[3:0]	•	Kernel	Bit [4:1] of Source Operand
H	•	Kernel	Bit 0 of Source Operand (If set, ignore all other flags states)

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. 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 kernel mode or user mode. However, in the user mode, KFLAG behaves as a FLAG instruction and only modifies bits Z, N, C, and V.



Note

Using the KFLAG instruction to place the processor in Exception or Interrupt mode does not alter the zero-overhead loop disable bit, L. The L bit is set automatically only when a genuine exception or interrupt causes the processor to enter Exception or Interrupt modes, or when this bit is restored by an RTIE instruction.

Pseudo Code

```

if ((src[0] == 1) && (STATUS32[7] == 0)) then                                     KFLAG
    STATUS32[0] = 1
    Halt()
else
    STATUS32[11:8] = src[11:8]
    if (STATUS32[7] == 0 && HAS_INTERRUPTS > 0) /* in kernel mode */
        STATUS32[5:1] = src[5:1]
        if (DIV_REM_OPTION == 1)
            STATUS32[13] = src[13]
        if (STACK_CHECKING == 1)
            STATUS32[14] = src[14]
        if (RGF_NUM_BANKS > 1)
            STATUS32[b:16] = src[b:16] /* b = 15+log2(RGF_NUM_BANKS) */
        if (LL64_OPTION == 1)
            STATUS32[19] = src[19]
    STATUS32[20] = src[20]
    STATUS32[31] = src[31]

```

Assembly Code Example

```
KFLAG 0x01          ; Halt processor (other flags not updated)
KFLAG 0x10          ; Set interrupt priority threshold to 8.
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
KFLAG	c	00100RRR001010011RRRCCCCCRRRRRR
KFLAG	u6	00100RRR011010011RRRuuuuuuRRRRRR
KFLAG	s12	00100RRR101010011RRRssssssSSSSSS
KFLAG<.cc>	c	00100RRR111010011RRRCCCCC0QQQQQ
KFLAG<.cc>	u6	00100RRR111010011RRRuuuuuu1QQQQQ

LD LDH LDW LDB LDD

Function

Load from Memory



Note

The LDW mnemonic is deprecated, and equivalent to LDH

Extension Group

LD, LDH, LDB: BASELINE

LDD: LL64_OPTION

LD_S R0-3,[h,u5]: CODE_DENSITY

LD_S.AS a,[b,c]: CODE_DENSITY

LD_S R1,[GP, s11]: CODE_DENSITY

Operation

$\text{addr} = \text{b} + \text{c}; \text{a} = * \text{addr};$

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 6-14 on page 268 lists the data size modes.

Data can be sign-extended by enabling sign extend `<.x>`. Table 6-15 on page 269 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 6-13 on page 268 lists the direct to memory bypass modes.

The address writeback 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 6-11 on page 267 lists the address writeback 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 instruction works with 64-bit operands, see “64-Bit Data”.

Using the sign extension suffix with an LDD instruction raises an Illegal Instruction exception.

Pseudo Code

```

if AA==0 then address = src1 + src2    /* LD */
if AA==1 then address = src1 + src2
if AA==2 then address = src1
if AA==3 and ZZ==0 then
    address = src1 + (src2 << 2)
if AA==3 and ZZ==2 then
    address = src1 + (src2 << 1)
if AA==3 and ZZ==3 then
    address = src1 + (src2 << 2)
if AA==1 or AA==2 then
    src1 = src1 + src2
DEBUG[LD] = 1

dest = Memory(address, size)           /* On Returning Load */
if X==1 then
    dest = Sign_Extend(dest, size)
if NoFurtherLoadsPending() then
    DEBUG[LD] = 0

```

Assembly Code Example

```
LD r0, [r1,4]    ; Load word from memory address r1+4 and write result to r0
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LD<zz><.x><.aa><.di>	a, [b]	00010bbb000000000BBBdaaZZXAAAAAA
LD<zz><.x><.aa><.di>	a, [b,s9]	00010bbbsssssssssSBBBdaaZZXAAAAAA
LD<zz><.x><.aa><.di>	a, [b,c]	00100bbbbaa110ZZXD BBBCCCCC AAAAAA
LD_S	a, [b,c]	01100bbbccc00aaa
LD_S	b, [SP,u7]	11000bbb000uuuuu
LD_S	b, [PCL,u10]	11010bbbuuuuuuuu
LD_S	c, [b,u7]	10000bbbcccuuuuu
LD_S	R0, [GP,s11]	1100100sssssssss
LDB_S	a, [b,c]	01100bbbccc01aaa
LDB_S	b, [SP,u7]	11000bbb001uuuuu

LDB_S	c, [b, u5]	10001bbbccccuuuuu
LDB_S	R0, [GP, s9]	1100101ssssssssss
LDH_S	a, [b, c]	01100bbbccc10aaa
LDH_S	c, [b, u6]	10010bbbccccuuuuu
LDH_S	R0, [GP, s10]	1100110ssssssssss
LDH_S.X	c, [b, u6]	10011bbbccccuuuuu

**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]	00100bbbaa110ZZXD BBBCCCCC111110

Encodings supported by the CODE DENSITY options

LD_S	R0, [h, u5]	01000U00hhhhuu1HH
LD_S	R1, [h, u5]	01000U01hhhhuu1HH
LD_S	R2, [h, u5]	01000U10hhhhuu1HH
LD_S	R3, [h, u5]	01000U11hhhhuu1HH
LD_S.AS	a, [b, c]	01001bbbccc00aaa
LD_S	R1, [GP, s11]	01010SSSSSS00sss

Encodings supported by LL64_OPTION**Note**

A represents a 64-bit operand, and should be specified as even numbered registers. For more information about 64-bit operands, see [“Data Formats”](#).

Instruction Code

LDD<.aa><.di>	a, [b]	00010bbb000000000BBBDaa110AAAAAA

LDD<.aa><.di>	a, [b, s9]	00010bbbssssssssSBBBdaa110AAAAAA
LDD<.aa><.di>	a, [b, c]	00100bbbaa110110DBBBCCCCCAAAAAA

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

LDI, LDI_S

Function

Load Indexed

Extension Group

CODE_DENSITY

Operation

$\text{addr} = \text{AUX\_LDI\_BASE} + (\text{src2} \ll 2); \text{b} = * \text{addr};$

Instruction Format

op b, [u7]

op b, [src2]

Syntax Example

LDI_S b, [u7]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

A memory address is computed as the sum of the AUX_LDI_BASE register and the scaled unsigned index operand, and the 32-bit word at that location in memory is read. The value read from memory is placed in the destination b register.

Pseudo Code

```
b = mem [AUX_LDI_BASE + (u7 << 2)]      /* LDI_S */
b = mem [AUX_LDI_BASE + (src2 << 2)]    /* LDI   */
```

Assembly Code Example

```
LDI_S r0, [45]    ; load into r0 from the location given by
                  ;AUX_LDI_BASE + 180
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
LDI	b, [c]	00100bbb00100110RBBBCCCCCRRRRRR
LDI	b, [u6]	00100bbb01100110RBBBuuuuuuRRRRRR
LDI	b, [s12]	00100bbb10100110RBBBssssssSSSSSS
LDI_S	b, u7	01010bbbUUUU1uuu

LEAVE_S

Function

Function epilog sequence

Extension Group

CODE_DENSITY

Operation

Pop a collection of registers from the stack, updating stack pointer and frame pointer, as required, and optionally jumping to the return address given by the contents of r31 (blink).

Instruction Format

op u7

Syntax Example

LEAVE_S {r13-r26, fp, blink, pcl}

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

This instruction performs the function epilog code, loading from the stack the set of registers defined by the 7-bit literal operand. On completion of this instruction, the stack pointer (r28) is incremented by 4S, where S is the total number of registers loaded.

For ENTER_S and LEAVE_S instructions, if the stack pointer is not 32-bit aligned, a misaligned access exception is always raised irrespective of the STATUS32.AD bit.

This instruction can optionally jump to the location given by the contents of the blink register, after all register loads are completed.

The status flags are not updated by this instruction.

The LEAVE_S instruction is not allowed to appear in the delay slot of a jump or branch instruction. The processor raises an Illegal Instruction Sequence exception on any attempt to execute LEAVE_S in a delay slot.



Note An illegal instruction sequence exception is not raised when the LEAVE_S instruction has zero operands.

The u7 encoding is as follows:

- u[3:0] Indicates the number of general-purpose registers to be restored from the stack, starting from r13 and counting contiguously upwards. When 32 general-purpose registers are configured, at most 14 registers (r13 to r26) can be restored. When only 16 general-purpose registers are configured, at most 3 registers (r13 to r15) can be restored. Therefore, if u[3:0] > 14 in a 32-register configuration, or if u[3:0] > 3 in a 16-register configuration, an Illegal Instruction exception is raised.
- u[4] If set to 1, r28 (stack-pointer) is set to r27 (frame-pointer) before registers are restored from the stack.
The frame-pointer is also restored from the stack.
- u[5] If set to 1, the blink register (r31) is restored.
- u[6] If set to 1, jump to the location given by the contents of the blink registers (r31) on completion of all register loads.

This instruction may perform as many as 16 reads from memory. These behave identically to a normal load instruction, such as:

```
LD r31, [sp,+16]
```

Pseudo Code

```

if (RGF_NUM_REGS == 32)
    MaxRegs = 14
else
    MaxRegs = 3

if (u[3:0] > MaxRegs)
    RaiseException (IllegalInstruction)

if (InDelaySlot)
    RaiseException (IllegalInstructionSequence)

M = 0                // initial sp offset = 0
W = 4

if (u[4] == 1)        // if frame-pointer saved
    sp = r[27]         // then set sp = fp

if (u[5] == 1)        // restore blink?
{
    r31 = mem[sp + M]
    M += W
}

for (i = 0; i < u[3:0]; i++)
{
    r[i+13] = mem[sp + M] // restore GPR i+13
    M += W
}

if (u[4] == 1)        // restore frame pointer?
{
    r27 = mem[sp + M]
    M += W
}

sp = sp + M           // deallocate stack space

if (u[6] == 1)        // if return jump required
    PC = r31           // jump [blink]

```

Assembly Code Example

```

LEAVE_S {r13-r26, blink, pcl}    ;Pop registers r13-r26, and blink
                                   ;from the stack, updating stack
                                   ;pointer.
                                   ;Afterwards, jump to the return
                                   ;address restored to blink.

```


Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code

LEAVE_S u7 11000UUU110uuuu0

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 SCND 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.

The LLOCK instruction generates exclusive transactions (in AXI this means the command attribute AxLOCK is asserted). For more information, see the [Atomic Memory Operations](#) section.

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](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code

LLOCK<.di>	b, [c]	00100bbb00101111DBBBCCCCCC010000
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](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
LLOCKD<.di>	b, [c]	00100bbb00101111DBBBCCCCCC010010
LLOCKD<.di>	b, [u6]	00100bbb01101111DBBBuuuuuu010010

LP<cc>

Function

Loop Set Up

Extension Group

BASELINE

Operation

if (!cc) PC = PCL + u7 else {LP_END = PCL + u7; LP_START = NEXT_PC;}

Instruction Format

op u7

op s13

Syntax Example

LP<cc> u7

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

This instruction sets up a zero-overhead loop that may be conditional or unconditional, and has a single literal operand which specifies the displacement to the end of the loop.

The literal operand (u7 or s13) specifies a relative displacement value that is added to the current 32-bit word-aligned PC value (PCL) to determine the loop-end address. This address represents the address of the first instruction after the zero-overhead loop body.

The number of iterations through the loop is nominally set by the LP_COUNT register. If LP_COUNT is set to 1, the loop body executes once. The LP_COUNT value is tested at the end of the loop for the loop termination value of 0x1, decremented before the next iteration. If the LP_COUNT register value is zero when the loop begins, the value at the end of the 1st iteration is decremented to 0xFFFF_FFFF (wraps), and the loop body is executed $(2^{32} - 1)$ times. Loops may be exited early by branch or jump instructions within the loop.

If a condition is specified by the LPcc instruction and the condition is *not* met, the program execution is resumed from the 16-bit aligned loop-end address.

If the condition is met, or if the instruction does not specify a condition, the auxiliary register LP_END (auxiliary register 0x03) is assigned the loop-end address. In parallel, the LP_START register (auxiliary register 0x02) is assigned the address of the next instruction after LPcc. These assignments take effect on completion of the LPcc instruction.

The non-conditional LP instruction always updates LP_END and LP_START auxiliary registers.

The conditional execution is summarized in the following table:

Loop Format	LPcc Semantics (Conditional Execution <cc>)	
	True	False
LPcc u7	$\text{aux\_reg}[\text{LP\_END}] = \text{PCL} + \text{u7}$ $\text{aux\_reg}[\text{LP\_START}] = \text{NEXT\_PC}$	$\text{PC} = \text{PCL} + \text{u7}$
LP s13	$\text{aux\_reg}[\text{LP\_END}] = \text{PCL} + \text{s13}$ $\text{aux\_reg}[\text{LP\_START}] = \text{NEXT\_PC}$	Not possible

Table 6-7 on page 264 lists the condition codes, cc.



Caution

The LPcc instruction must not be in the executed delay slot of branch or jump instructions, and therefore, the LPcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBitn.D instruction.

Because LP_END is set to 0 upon [Reset](#), you must not execute an instruction placed at the end of program memory space (0xFFFFF0FC or 0xFFFFF0FE) because such instructions trigger the LP mechanism if no other LP has been set up since [Reset](#).

Use caution if code is copied or overlaid into memory or if using any form of MMU or memory mapping; Ensure before executing the code that LP_END is initialized to a safe value (0) to prevent accidental LP triggering.

Care should be taken to align LP_END with an instruction boundary position. An Illegal Instruction Sequence exception is raised when LP_END is at 16-bit offset position immediately after an RTIE (32-bit) instruction.

The LP instruction is encoded to use immediate values (syntax u7 or syntax s13). Encoding the operand mode (bits 23:22) to be 0x0 is illegal and raises an Illegal Instruction exception. In addition, using operand mode 0x3 with sub-operand mode 0x0 is illegal, and also raises an Illegal Instruction exception. The processor ignores the reserved field, R.

The LP instruction may be used to set up a loop with a maximum loop size determined by the limit of the branch offset available in each of the two LP encodings.

- ❑ Conditional LPcc – 6 bits of unsigned offset
- ❑ Unconditional LP – 12 bits of signed offset

In common with all relative branch offsets, the calculation of the LP branch target, which defines the first instruction after the loop, is given by the sum of the current 32-bit word-aligned PC (cPCL) plus

the offset. The offset is encoded as a half-word displacement, and is therefore scaled by a 1-bit left shift before being added to cPCL.

A 6-bit unsigned offset therefore provides a maximum displacement of 126 bytes from a 32-bit word-aligned LP instruction, or 124 bytes from an LP instruction at aligned to an odd half-word address. The LPcc instruction itself occupies the first 4 bytes of this displacement, and therefore the maximum loop size for a conditional LPcc instruction is 122 or 120 bytes for 32-bit aligned and 16-bit aligned LPcc instructions respectively.

The unconditional form of the LP instruction has a signed 12-bit offset, allowing an effective displacement range of -4096 to +4094 bytes for a 32-bit aligned LP, and -4098 to +4092 bytes for a 16-bit aligned LP instruction.

Jumps and control-transfer instructions, and their delay-slots where present, may not appear as the last instruction of a zero-overhead loop body.

One zero-overhead loop may be used inside another if the inner loop saves and restores the context of the outer loop and complies with all other rules. An additional rule is that a loop instruction may not appear in the last position, or the penultimate position, of a zero-overhead loop body.

The following instructions are prohibited in the last position (the instruction immediately before LOOP_END) of a zero-overhead loop body:

- ❑ A branch or jump instruction (with or without a delay slot and either taken or not taken) (B, Bcc, Bcc_S, BR, BRcc, BRcc_S, BBIT0, BBIT1, BL, BLcc, BL_S, BI, BIH, J, Jcc, J_S, Jcc_s, J blink, JLcc, JL_S)
- ❑ The delay-slot instruction of a taken branch or jump.



Note

If a branch is not taken it is not always possible to know that the delay-slot instruction is actually a delay slot. That is the case if there is an intervening exception.

- ❑ ENTER_S, LEAVE_S, RTIE
- ❑ EI_S
- ❑ E-slot (the execute slot of an EI_S instruction).
- ❑ JLI_S
- ❑ LPcc

If the prohibited instructions are used in the last position of a zero-overhead loop, the effect is implementation dependent. For further details, see the Databook of each ARCV2-based processor.

The use of zero delay loops is illustrated in the following example.

Example 9-3 Example Loop Code

```

MOV LP_COUNT,2      ; do loop 2 times (flags not set)
...                ; Some intermediate instructions
LP loop_end         ; set up loop mechanism to work
                   ; between loop_in and loop_end
loop_in: LR r0,[r1]  ; first instruction in loop
ADD r2,r2,r0        ; sum r0 with r2
BIC r1,r1,4         ; last instruction
                   ; in loop
loop_end:
ADD r19,r19,r20     ; first instruction after loop

```

Direct writes to the LP_START and LP_END registers can be used to set up larger loops, if required.

**Caution**

When directly manipulating LP_START and LP_END, ensure that the values written refer to the first address occupied by an instruction.

Loop Mechanism

The loop mechanism is inactive whenever the loop-inhibit bit STATUS32[L] is set to one. This bit is set, thereby disabling the loop mechanism on an interrupt or an exception (including TRAP instructions). Loops are enabled (STATUS32[L]=0) after [Reset](#). The loop-inhibit bit is cleared (loops allowed) whenever the processor commits a taken conditional LPcc instruction or an unconditional LP instruction. From kernel mode, the value of the bit can also be restored using the RTIE instruction.

When the loop mechanism is disabled (STATUS32[L]=1), loop-end conditions are ignored, no change of program flow takes place, and the LP_COUNT register is not decremented. The STATUS32[L] register does not affect the reads and writes to the loop control registers.

The processor checks for a loop-end condition when calculating the next program counter address, before each instruction is completed.

A loop-end condition is an implicit branch to the start of the loop (LP_START). A loop-end condition is detected when **all** of the following conditions are met:

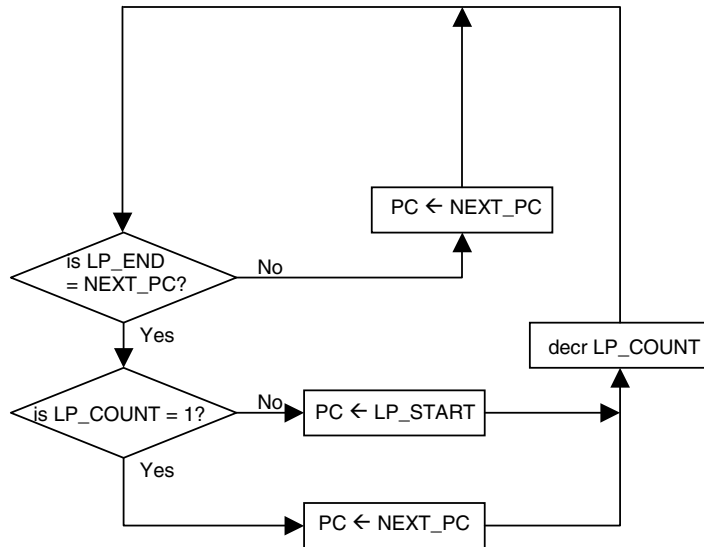
- ❑ STATUS32[DE] is 0, indicating that the instruction is not in the delay slot of a branch, or the branch was not taken.
- ❑ STATUS32[L] is 0
 - This bit is set to 1 to disable loop-end detection.
- ❑ The instruction to be completed is the last in a loop
 - $PC + \text{current instruction size} = LP\_END$
- ❑ LP_COUNT is not equal to 1
 - When LP_COUNT=1, LP_COUNT is decremented and execution continues from the instruction pointed to by LP_END.
- ❑ $(LP\_END - LP\_START) \geq 4$

- If this condition is not true, LP_START and LP_END define a loop body containing fewer than 4 bytes. This size is smaller than the minimum supported zero-overhead loop size in the ARCV2 architecture.

When a loop-end condition is detected, the processor jumps to the address in LP_START, and LP_COUNT is decremented.

If LP_COUNT is 1, the processor continues execution from the instruction pointed to by LP_END; LP_COUNT is also decremented. This sequence is illustrated in [Figure 9-1](#).

Figure 9-1 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 9-3](#). This table covers loop setup and use of long immediate data.



Instruction numbers islotn refer to the sequence of executed instructions within a loop, which is not the same as the static instruction positions in the code, if branches are used within the loop.

Instruction numbers oslotn refer to instructions outside the loop.

Table 9-3 When Reads and Writes Take Effect

	Instruction	Writing Takes Effect	Reading Uses	Writing Takes Effect	Reading Uses
Loop_start:		LP_COUNT	LP_COUNT	LP_END, LP_START	LP_END, LP_START
	islot1	current iteration	current iteration	current iteration	current iteration
	islot2	current iteration	current iteration	current iteration	current iteration
	islotn-3	current iteration	current iteration	current iteration	current iteration
	islotn-2	current iteration	current iteration	current iteration	current iteration
	islotn-1	current iteration	current iteration	current iteration	current iteration
	islotn	next cycle	current iteration	next cycle	current iteration
Loop_end:					
	oslot1				
	oslot2				

Pseudo Code

```

if cc==true then                                /* LPcc */
  Aux_reg(LP_START) = nPC
  Aux_reg(LP_END) = cPCL + rd
  PC = nPC
else
  PC = cPCL + rd

```

Assembly Code Example

```

LPNE label                                     ; if the Z flag is set then
                                                ; branch to label else
                                                ; set LP_START to address of
                                                ; next instruction and set
                                                ; LP_END to label

```

The use of zero delay loops is illustrated in [Example 9-4](#).

Example 9-4 Use of Zero Overhead Loop

```

        MOV LP_COUNT,2    ; do loop 2 times (flags not set)
        LP loop_end      ; set up loop mechanism to work
                           ; between loop_in and loop_end
loop_in: LR r0,[r1]       ; first instruction of loop
        ADD r2,r2,r0      ; second instruction of loop
        BIC r1,r1,4       ; last instruction of loop
loop_end:
        ADD r19,r19,r20   ; first instruction after loop

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LP<cc>	u7	00100RRR11101000RRRRuuuuuu1QQQQQ
LP	u7	00100RRR01101000RRRRuuuuuuRRRRRR
LP	s13	00100RRR10101000RRRRssssssSSSSSS

LR

Function

Load from Auxiliary Register

Extension Group

BASELINE

Operation

$b = \_lr(c);$

Instruction Format

op b,c

Syntax Example

LR b,[c]

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Get the data from the auxiliary register whose number is obtained from the source operand (c) and place the data into the destination register (b).

The LR instruction executes unconditionally. Therefore, specifying an operand mode (bits 23:22) of 0x3 raises an [Illegal Instruction](#).

Pseudo Code

```
dest = Aux_reg(src)          /* LR */
```

Assembly Code Example

```
LR r1, [r2]      ; Load contents of Aux. register ;pointed
                  ; to by r2 into r1
```

Syntax and Encoding

The status flags are not updated with this instruction therefore the flag setting field, F, is encoded as 0. The reserved field, R, is ignored by the processor, but must be set to 0.

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LR	b, [c]	00100bbb00101010RBBBCCCCCRRRRRR
LR	b, [u6]	00100bbb01101010RBBBuuuuuuRRRRRR
LR	b, [s12]	00100bbb10101010RBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

LSL16

Function

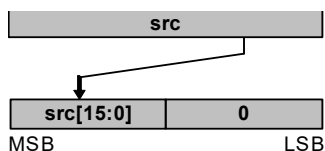
Logical Shift Left 16

Extension Group

SWAP_OPTION

Operation

$b = c \ll 16;$



Instruction Format

op b,c

Syntax Example

LSL16<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 16 places to the left, clearing the lower 16 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src << 16                                     /* LSL16 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
LSL16 r1,r2           ; Logical shift of r2 16
                      ;places to the
                      ; left, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

LSL16<.f>	b, c	00101	bbb	00	101111	FBBB	CCCCC	001010
LSL16<.f>	b, u6	00101	bbb	01	101111	FBBB	uuuuuu	001010

LSL8

Function

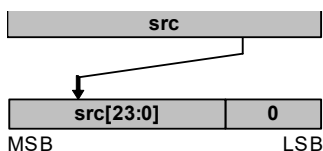
Logical Shift Left 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$b = c \ll 8;$



Instruction Format

op b,c

Syntax Example

LSL8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 8 places to the left, clearing the lower 8 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src << 8                                     /* LSL8 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
LSR8 r1,r2      ; Logical shift of r2 8 places to the left, placing
                ;result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

LSL8<.f>	b,c	00101	bbb00	101111	FBBB	CCCCC	001111
LSL8<.f>	b,u6	00101	bbb01	101111	FBBB	uuuuuu	001111

LSR

Function

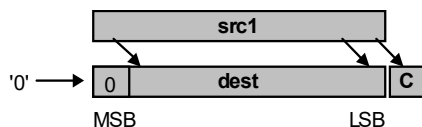
Logical Shift Right

Extension Group

BASELINE

Operation

$b = (\text{unsigned})\ c \gg 1;$



Instruction Format

op b,c

Syntax Example

LSR<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Logically right shift the source operand (c) by one and place the result into the destination register (b).

The most-significant bit of the result is replaced with 0.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

dest = src >> 1                                /* LSR */
dest[31] = 0
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[0]

```

Assembly Code Example

```

LSR r1,r2    ; Logical shift right contents of r2 by one bit and write
              ;result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LSR<.f>	b, c	00100bbb00101111FBBBCCCCC000010
LSR<.f>	b, u6	00100bbb01101111FBBBuuuuuu000010
LSR_S	b, c	01111bbbccc11101

LSR multiple

Function

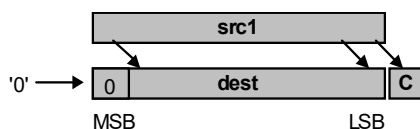
Multiple Logical Shift Right

Extension Group

SHIFT_OPTION 2 or 3

Operation

if (cc) a = (unsigned) b >> c;



Instruction Format

op a, b, c

Syntax Example

LSR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Logically shift right b by c places and place the result in the destination register. Only the bottom 5 bits of c are used as the shift value.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* LSR
dest = src1 >> (src2 & 31)                      Multiple */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = if src2==0 then 0 else src1[src2-1]

```

Assembly Code Example

```

LSR r1,r2,r3    ; Logical shift right contents of r2 by r3 bits and
                ;write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

LSR<.f>	a,b,c	00101bbb0000001FBBBCCCCCAAAAAA
LSR<.f>	a,b,u6	00101bbb01000001FBBBuuuuuuAAAAAA
LSR<.f>	b,b,s12	00101bbb1000001FBBBssssssSSSSSS
LSR<.cc><.f>	b,b,c	00101bbb11000001FBBBCCCCC0QQQQQ
LSR<.cc><.f>	b,b,u6	00101bbb11000001FBBBuuuuuu1QQQQQ
LSR_S	b,b,c	01111bbbccc11001
LSR_S	b,b,u5	10111bbb001uuuuu

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

LSR16

Function

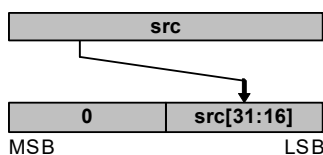
Logical Shift Right 16

Extension Group

SWAP_OPTION

Operation

$b = c \gg 16;$



Instruction Format

op b,c

Syntax Example

LSR16<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= 0
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 16 places to the right, clearing the upper 16 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 16                                /* LSR16 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```


Assembly Code Example

```
LSR16 r1,r2    ; Logical shift of r2 16 places to the right, placing
                ;result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

LSR16<.f>	b, c	00101bbb00101111FBBBCCCCC001011
LSR16<.f>	b, u6	00101bbb01101111FBBBuuuuuu001011

LSR8

Function

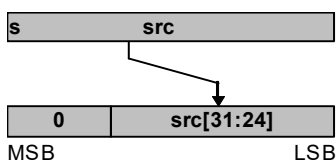
Logical Shift Right 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$b = c \gg 8;$



Instruction Format

op b,c

Syntax Example

LSR8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= 0
C		= Unchanged
V		= Unchanged

Description

Shift the source operand 8 places to the right, clearing the upper 8 bits.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 8                                     /* LSR8 */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
LSR8 r1,r2    ; Logical shift of r2 8 places to the
               ; right, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

LSR8<.f>	b,c	00101bbb00101111FBBBCCCCC001110
LSR8<.f>	b,u6	00101bbb01101111FBBBuuuuuu001110

MAC

This section describes the standard MAC instruction. For the DSP version, see [MAC](#).

Function

Signed 32x32 multiplication and accumulation.

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    a = result.w0;
    acc = result;
}
```



Note

w0 represents the lower 32-bits of a 64-bit result. For more information about 64-bit operands, see [“Data Formats”](#).

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

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
MAC<.f>	a,b,c	00101bbb00001110FBBBCCCCCAAAAA
MAC<.f>	a,b,u6	00101bbb01001110FBBBuuuuuuAAAAAA
MAC<.f>	b,b,s12	00101bbb10001110FBBBsssssssSSSSSS
MAC<.cc><.f>	b,b,c	00101bbb11001110FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MACD

This section describes the standard MACD instruction. For the DSP version, see [MACD](#).

Function

Signed 32x32 multiplication and accumulation. Returns a 64-bit result.

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    A = result;
    acc = result;
}
```



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”](#).

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

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
MACD<.f>	a,b,c	00101bbb00011010FBBBCCCCCAAAAAA
MACD<.f>	a,b,u6	00101bbb01011010FBBBuuuuuuAAAAAA
MACD<.f>	b,b,s12	00101bbb10011010FBBBssssssSSSSSS
MACD<.cc><.f>	b,b,c	00101bbb11011010FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MACDU

This section describes the standard MACDU instruction. For the DSP version, see [MACDU](#).

Function

Unsigned 32x32 multiplication and accumulation. Return a 64-bit result.

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (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”](#).

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) {                                     /* MACDU*/
    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



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MACDU<.f>	a,b,c	00101bbb00011011FBBBCCCCCAAAAA
MACDU<.f>	a,b,u6	00101bbb01011011FBBBuuuuuuAAAAA
MACDU<.f>	b,b,s12	00101bbb10011011FBBBssssssSSSSSS
MACDU<.cc><.f>	b,b,c	00101bbb11011011FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MACU

Function

Unsigned 32x32 multiplication and accumulation. For the DSP version, see [MACU](#).

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    a = result.w0;
    acc = result;
}
```

 **Note**

w0 represents the lower 32-bits of a 64-bit result.

Instruction Format

op a, b, c

Syntax Example

MACU <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V	•	= Set if the accumulator overflows. This instruction never clears this flag.

Description

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) {                                                    /* MACU*/
    result = acc + (b * c);
    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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
MACU<.f>	a,b,c	00101bbb00001111FBBBCCCCC AAAAAA
MACU<.f>	a,b,u6	00101bbb01001111FBBBuuuuuu AAAAAA
MACU<.f>	b,b,s12	00101bbb10001111FBBBssssss SSSSSS
MACU<.cc><.f>	b,b,c	00101bbb11001111FBBBCCCCC 0QQQQQ
MACU<.cc><.f>	b,b,u6	00101bbb11001111FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MAX

Function

Return Maximum Value

Extension Group

BASELINE

Operation

if (cc) a = (b>c) ? b : c;

Instruction Format

op a, b, c

Syntax Example

MAX<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if both source operands are equal
N	•	= Set if most-significant bit of result of src1-src2 is set
C	•	= Set if src2 is selected (src2 >= src1)
V	•	= Set if an overflow is generated (as a result of src1-src2)

Description

Return the maximum of the two signed source operands (b and c) and place the result in the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* MAX */
    alu = src1 - src2
    if src2 >= src1 then
        dest = src2
    else
        dest = src1
    if F==1 then
        Z_flag = if alu==0 then 1 else 0
        N_flag = alu[31]
        V_flag = Overflow()
        C_flag = if src2>=src1 then 1 else 0

```

Assembly Code Example

```
MAX r1,r2,r3    ; Take maximum of r2 and r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MAX<.f>	a,b,c	00100bbb00001000FBBBCCCCC	AAAAAA
MAX<.f>	a,b,u6	00100bbb01001000FBBBuuuuuu	AAAAAA
MAX<.f>	b,b,s12	00100bbb10001000FBBBssssss	SSSSSS
MAX<.cc><.f>	b,b,c	00100bbb11001000FBBBCCCCC	0QQQQQ
MAX<.cc><.f>	b,b,u6	00100bbb11001000FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MIN

Function

Return Minimum Value

Extension Group

BASELINE

Operation

if (cc) a = (b < c) ? b : c;

Instruction Format

op a, b, c

Syntax Example

MIN<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if both source operands are equal
N	•	= Set if most-significant bit of result of src1-src2 is set
C	•	= Set if src2 is selected (src2 <= src1)
V	•	= Set if an overflow is generated (as a result of src1-src2)

Description

Return the minimum of the two signed source operands (b and c) and place the result in the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* MIN */
    alu = src1 - src2
    if src2 <= src1 then
        dest = src2
    else
        dest = src1
    if F==1 then
        Z_flag = if alu==0 then 1 else 0
        N_flag = alu[31]
        V_flag = Overflow()
        C_flag = if src2<=src1 then 1 else 0

```

Assembly Code Example

```
MIN r1,r2,r3    ; Take minimum of r2 and r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MIN<.f>	a,b,c	00100bbb00001001FBBBCCCCC	AAAAAA
MIN<.f>	a,b,u6	00100bbb01001001FBBBuuuuuu	AAAAAA
MIN<.f>	b,b,s12	00100bbb10001001FBBBssssss	SSSSSS
MIN<.cc><.f>	b,b,c	00100bbb11001001FBBBCCCCC	0QQQQQ
MIN<.cc><.f>	b,b,u6	00100bbb11001001FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MOV

Function

Move Contents

Extension Group

BASELINE

Operation

if (cc) b = c;

Instruction Format

op b, c

Syntax Example

MOV<.cc><.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

The contents of the source operand (c) are moved to the destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then /* MOV */           /* MOV */
    dest = src
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
```

Assembly Code Example

```
MOV r1,r2      ; Move contents of r2 into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MOV<.f>	b,c	00100bbb00001010FBBBCCCCCRRRRRR
MOV<.f>	b,u6	00100bbb01001010FBBBuuuuuuRRRRRR
MOV<.f>	b,s12	00100bbb10001010FBBBssssssSSSSSS
MOV<.cc><.f>	b,c	00100bbb11001010FBBBCCCCC0QQQQQ
MOV<.cc><.f>	b,u6	00100bbb11001010FBBBuuuuuu1QQQQQ
MOV_S	h,s3	01110ssshhh011HH
MOV_S	0,s3	01110sss11001111
MOV_S.NE	b,h	01110bbbh hh111HH
MOV_S.NE	b,limm	01110bbb11011111
MOV_S	b,u8	11011bbbuuuuuuuu
MOV_S	g,h	01000ggghhhGG0HH
MOV_S	g,limm	01000ggg110GG011
MOV_S	0,h	01000110hhh110HH
MOV_S	0,limm	0100011011011011

MPY, MPY_S

Function

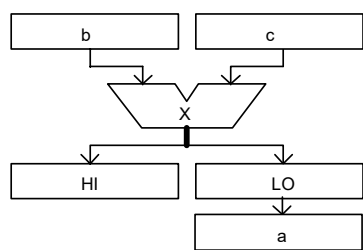
32 x 32 Signed Multiply, returning least-significant 32-bit word of the result. For the DSP version, see [MPY](#).

Extension Group

(MPY_OPTION > 1) or (HAS_DSP == 1)

Operation

if (cc) a = (signed) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPY<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 64-bit result is set
C		= Unchanged
V	•	= Set when the signed result cannot be wholly contained within the lower part of the 64-bit result. In other words, when bits 62:31 do not equal bit 64, the sign bit.

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination operand register.

Pseudo Code

```
if cc==true then                                     /* MPY */
    dest = (src1 * src2) & 0x0000_0000_FFFF_FFFF
```

Assembly Code Example

```
MPY r1,r2,r3          ; Multiply r2 by r3 and put the least
                      ; significant word of the result in r1
MPY_S r1,r2           ; 16-bit encoding of MPY r1,r1,r2
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
MPY<.f>	a,b,c	00100bbb00011010FBBBCCCCCAAAAAA
MPY<.f>	a,b,u6	00100bbb01011010FBBBuuuuuuAAAAAA
MPY<.f>	b,b,s12	00100bbb10011010FBBBssssssSSSSSS
MPY<.cc><.f>	b,b,c	00100bbb11011010FBBBCCCCC0QQQQQ
MPY<.cc><.f>	b,b,u6	00100bbb11011010FBBBuuuuuu1QQQQQ
MPY_S	b,b,c	01111bbbccc01100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYD

This section describes the standard MPYD instruction. For the DSP version, see [MPYD](#).

Function

Signed 32x32 multiplication. Return a 64-bit result.

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```



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”](#).

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) {
    result = (b * c);
    A = result;
    acc = result;
}
```

/* MPYD*/

Assembly Code Example

```
MPYD r0,r2,r3    ; 32x32 multiplication of r2 and r3. The result is
                  ; stored in (r1, r0).
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
MPYD<.f>	a,b,c	00101bbb000111000FBBBCCCCC AAAAAA
MPYD<.f>	a,b,u6	00101bbb010111000FBBBuuuuuu AAAAAA
MPYD<.f>	b,b,s12	00101bbb100111000FBBBssssss SSSSSS
MPYD<.cc><.f>	b,b,c	00101bbb110111000FBBBCCCCC 0QQQQQ
MPYD<.cc><.f>	b,b,u6	00101bbb110111000FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYDU

Function

Unsigned 32x32 multiplication with a 64-bit result. For the DSP version, see [MPYDU](#).

Extension Group

MPY_OPTION > 7 or has_dsp==true

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"](#).

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



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MPYDU<.f>	a,b,c	00101bbb00011001FBBBCCCCCAAAAA
MPYDU<.f>	a,b,u6	00101bbb01011001FBBBuuuuuuAAAAA
MPYDU<.f>	b,b,s12	00101bbb10011001FBBBssssssSSSSSS
MPYDU<.cc><.f>	b,b,c	00101bbb11011001FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYM MPYH

Function

32 x 32 Signed Multiply High. For the DSP version, see [MPYM MPYH](#)

 **Note**

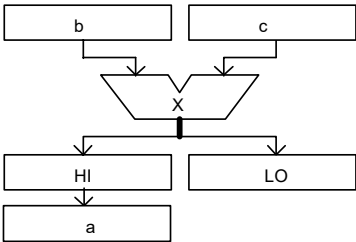
The MPYH mnemonic is deprecated.

Extension Group

(MPY_OPTION > 1) or (HAS_DSP == 1)

Operation

if (cc) a = (signed) (b * c) >> 32;



Instruction Format

op a, b, c

Syntax Example

MPYM<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set is the result is negative
C		= Unchanged
V	•	= Always cleared.

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2, placing the most-significant 32 bits of the 64-bit result in the destination register. Any flag updates occur only if the set

flags suffix (.F) is used. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if cc==true then                                     /*MPYM*/
    dest = (src1 * src2) >> 32
```

Assembly Code Example

```
MPYM r1,r2,r3      ;Multiply r2 by r3 and put high part of the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MPYM<.f>	a,b,c	00100bbb00011011FBBBCCCCCAAAAA
MPYM<.f>	a,b,u6	00100bbb01011011FBBBuuuuuuAAAAA
MPYM<.f>	b,b,s12	00100bbb10011011FBBBssssssSSSSSS
MPYM<.cc><.f>	b,b,c	00100bbb11011011FBBBCCCCC0QQQQQ
MPYM<.cc><.f>	b,b,u6	00100bbb11011011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYMU, MPYHU

Function

32 x 32 Unsigned integer Multiply High. For the DSP version, see [MPYMU MPYHU](#)

 **Note**

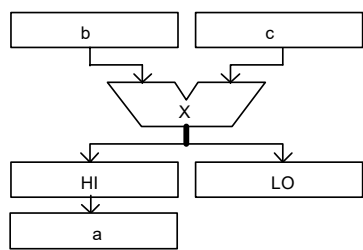
The MPYHU mnemonic is deprecated.

Extension Group

(MPY_OPTION > 1) or (HAS_DSP == 1)

Operation

if (cc) a = (unsigned) (b * c) >> 32;dest (src1 X src2).high



Instruction Format

op a, b, c

Syntax Example

MPYMU<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared.
C		= Unchanged
V	•	= Always cleared.

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand 1 and operand 2, placing the most-significant 32 bits of the 64-bit result in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used. Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if cc==true then                                /*MPYMU*/
    dest = (src1 * src2) >> 32                 /*when MPY_OPTION==2
```

Assembly Code Example

```
MPYMU r1,r2,r3      ;Multiply r2 by r3 and put high part of the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

MPYMU<.f>	a,b,c	00100bbb00011100FBBBCCCCCAAAAAA
MPYMU<.f>	a,b,u6	00100bbb01011100FBBBuuuuuuAAAAAA
MPYMU<.f>	b,b,s12	00100bbb10011100FBBBssssssSSSSSS
MPYMU<.cc><.f>	b,b,c	00100bbb11011100FBBBCCCCC0QQQQQ
MPYMU<.cc><.f>	b,b,u6	00100bbb11011100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYU

Function

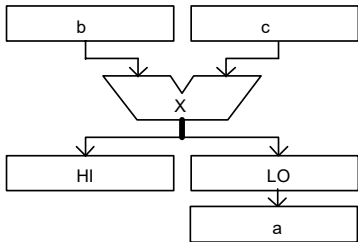
32 x 32 Unsigned Multiply, returning least-significant 32-bit word of result. For the DSP version, see [MPYU](#).

Extension Group

(MPY_OPTION > 1) or (HAS_DSP == 1)

Operation

if (cc) a = (unsigned) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPYU<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Set when the result cannot be represented as a 32-bit unsigned integer

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand1 and operand2, placing the least-significant 32 bits of the 64-bit result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if cc==true then                                     /* MPYU */
    dest = (src1 * src2) & 0x0000_0000_FFFF_FFFF
```

Assembly Code Example

```
MPYU r1,r2,r3    ;Multiply r2 by r3 and put low part of the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
MPYU<.f>	a,b,c	00100bbb000111101FBBBCCCCCAAAAAA
MPYU<.f>	a,b,u6	00100bbb010111101FBBBuuuuuuAAAAAA
MPYU<.f>	b,b,s12	00100bbb100111101FBBBssssssSSSSSS
MPYU<.cc><.f>	b,b,c	00100bbb110111101FBBBCCCCC0QQQQQ
MPYU<.cc><.f>	b,b,u6	00100bbb110111101FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYUW, MPYUW_S

Function

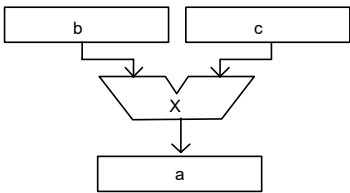
16 x 16 unsigned Multiply, return 32-bit word result. For the DSP version, see [MPYUW](#).

Extension Group

(MPY_OPTION > 0) or (HAS_DSP == 1)

Operation

if (cc) a = (unsigned) ((word) b * (word) c)



Instruction Format

op a, b, c

Syntax Example

MPYUW<.f> a,b,c
MPYUW_S b,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Always cleared

Description

Perform an unsigned 16-bit by 16-bit multiply of operand 1 and operand 2, and place 32-bit result in the destination register. The 16-bit source operands use the lower 16 bits of the source operands.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                     /*MPYUW*/
    dest = src1 * src2
```

Assembly Code Example

```
MPYUW r1,r2, r3          ; Multiply r2 by r3 and put the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
MPYUW<.f>	a,b,c	00100bbb00011111FBBBCCCCCAAAAAA
MPYUW<.f>	a,b,u6	00100bbb01011111FBBBuuuuuuAAAAAA
MPYUW<.f>	b,b,s12	00100bbb10011111FBBBssssssSSSSSS
MPYUW<.cc><.f>	b,b,c	00100bbb11011111FBBBCCCCC0QQQQQ
MPYUW<.cc><.f>	b,b,u6	00100bbb11011111FBBBuuuuuu1QQQQQ
MPYUW_S	b,b,c	01111bbbccc01010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

MPYW, MPYW_S

Function

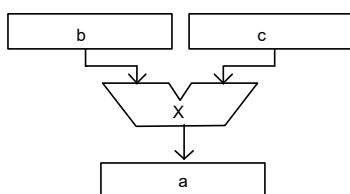
16 x 16 Signed Multiply. For the DSP version, see [MPYW](#).

Extension Group

(MPY_OPTION > 0) or (HAS_DSP == 1)

Operation

if (cc) a = (signed) ((word) b * (word) c)



Instruction Format

op a, b, c

Syntax Example

MPYW<.f> a,b,c

MPYW_S b,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 32-bit result is set
C		= Unchanged
V	•	= Set when the unsigned result overflows

Description

Perform a signed 16-bit by 16-bit multiply of operand 1 and operand 2, and place the 32-bit result in the destination register. The 16-bit source operands use the lower 16 bits of the source operands.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                /* MPYW */
    dest = src1 * src2
```

Assembly Code Example

```
MPYW r1,r2, r3          ; Multiply r2 by r3 and put the result in r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
MPYW<.f>	a,b,c	00100bbb00011110FBBBCCCCCAAAAAA
MPYW<.f>	a,b,u6	00100bbb01011110FBBBuuuuuuAAAAAA
MPYW<.f>	b,b,s12	00100bbb10011110FBBBssssssSSSSSS
MPYW<.cc><.f>	b,b,c	00100bbb11011110FBBBCCCCC0QQQQQ
MPYW<.cc><.f>	b,b,u6	00100bbb11011110FBBBuuuuuu1QQQQQ
MPYW_S	b,b,c	01111bbbccc01001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

NEG

Function

Negate

Extension Group

BASELINE

Operation

if (cc) $a = 0 - b$;

Instruction Format

op a,b

Syntax Example

NEG<.f> a,b

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

The negate instruction subtracts the source operand (b) from zero and places the result into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* NEG */
    dest = 0 - src1
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
NEG r1,r2      ; Negate r2 and write result into r1
```

Syntax and Encoding

The 32-bit instruction format is an encoding of the reverse subtract instruction, [RSUB](#), using an unsigned 6-bit immediate value set to 0.

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
NEG<.f>	a,b	00100bbb01001110FBBB000000AAAAAA
NEG<.cc><.f>	b,b	00100bbb11001110FBBB0000001QQQQQ
NEG_S	b,c	01111bbbccc10011

NOP

Function

No Operation

Extension Group

BASELINE

Operation

No Operation

Instruction Format

op

Syntax Example

NOP

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

No operation. The processor advances to the next instruction.

Pseudo Code

```
/* NOP_S */
```

Assembly Code Example

```
NOP_S ; No operation
```

Syntax and Encoding

The 32-bit NOP is an encoding of the MOV instruction (syntax MOV 0,u6) using the [General Operations Register with Unsigned 6-bit Immediate](#) format on [page 329](#). For more information about the encodings, see

“Key for 32-bit Addressing Modes and Encoding Conventions” and “Key for 16-bit Addressing Modes and Encoding Conventions” .

Instruction Code

NOP_S	0111100011100000
NOP	00100110010010100111000000000000

NORM

Function

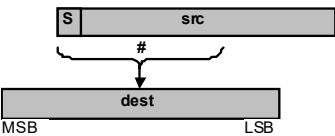
Normalize

Extension Group

BITSCAN_OPTION

Operation

```
for (i=0; (i<=31) && (c>>i != 0) && (c>>i != -1); i++); b= 31-i;
```



Instruction Format

op b,c

Syntax Example

NORM<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Computes the normalization integer for the signed value in the operand c. The normalization integer is the amount by which the operand must be shifted left to normalize the operand as a 32-bit signed integer.

Any flag updates occur only if the set flags suffix (.F) is used. The returned value for source operand of zero is 0x0000001F. Examples of returned values are shown in the following table:

Operand Value	Returned Value
0x00000000	0x0000001F
0x00000001	0x0000001E
0x1FFFFFFF	0x00000002
0x3FFFFFFF	0x00000001
0x7FFFFFFF	0x00000000
0x80000000	0x00000000
0xC0000000	0x00000001
0xE0000000	0x00000002
0xFFFFFFFF	0x0000001F

Pseudo Code

```
for (i=0;i<=31;i++)                /* NORM */
    if (src>>i ==0) break
    if (src>>i ==-1) break
end for
dest = 31-i
if F==1 then
    Z_flag = if src==0 then 1 else 0
    N_flag = src[31]
```

Assembly Code Example

```
NORM r1,r2    ; Normalization integer for r2, write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code

NORM<.f>	b, c	00101bbb00101111FBBBCCCCC000001
NORM<.f>	b, u6	00101bbb01101111FBBBuuuuuu000001

NORMH, NORMW

Function

Normalize 16-bit Half-word



Note

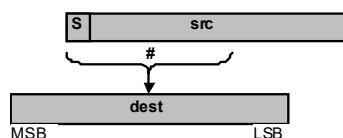
The NORMW mnemonic is deprecated.

Extension Group

BITSCAN_OPTION

Operation

```
for (i=0; (i<=15) && ((half-word) c>>i != 0) && ((half-word) c>>i != -1); i++);
b= 15-i;
```



Instruction Format

op b,c

Syntax Example

NORMH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Computes the normalization integer for the signed value in the operand. The normalization integer is the amount by which the operand must be shifted left to normalize the operand as a 16-bit signed integer. When normalizing a 16-bit signed integer, the lower 16 bits of the source data (c) is used.

Any flag updates occur only if the set flags suffix (.F) is used.

The returned value for source operand of zero is 0x000F. Examples of returned values are shown in the following table:

Operand Value	Returned Value
0x0000	0x000F
0x0001	0x000E
0x1FFF	0x0002
0x3FFF	0x0001
0x7FFF	0x0000
0x8000	0x0000
0xC000	0x0001
0xE000	0x0002
0xFFFF	0x000F

Pseudo Code

```
for (i=0;i<=15;i++)                                /* NORMW */
    if ((half-word)src>>i ==0) break
    if ((half-word)src>>i ==-1) break
end for
dest = 15-i
if F==1 then
    Z_flag = if (src & 0x0000FFFF)==0 then 1 else 0
    N_flag = src[15]
```

Assembly Code Example

```
NORMH r1,r2    ; Normalization integer for lower 16 bits of r2
               ; write result into r1
```

Syntax and Encoding

For more information about the encodings, see “Key for 32-bit Addressing Modes and Encoding Conventions” and “Key for 16-bit Addressing Modes and Encoding Conventions” .

Instruction Code

```
NORMH<.f>    b,c    00101bbb00101111FBBBCCCCC001000
```

NORMH<.f> b,u6 00101bbb01101111FBBBuuuuuu001000

NOT

Function

Bitwise NOT

Extension Group

BASELINE

Operation

$b = \sim c;$

Instruction Format

op b,c

Syntax Example

NOT<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Bitwise NOT (inversion) of the source operand (c) with the result placed into the destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = NOT(src)                                /* NOT */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
NOT r1,r2    ; Logical bitwise NOT r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
NOT<.f>	b, c	00100bbb00101111FBBBCCCCC001010
NOT<.f>	b, u6	00100bbb01101111FBBBuuuuuu001010
NOT_S	b, c	01111bbbccc10010

OR

Function

Bitwise OR

Extension Group

BASELINE

Operation

if (cc) $a = b \mid c$;

Instruction Format

op a, b, c

Syntax Example

OR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise OR of source operand 1 (b) with source operand 2 (c). The result is written into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* OR */
  dest = src1 OR src2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]

```


Assembly Code Example

```
OR r1,r2,r3    ; Logical bitwise OR contents of r2 with r3
               ; and write result into r1;
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
OR<.f>	a,b,c	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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

POP

Function

Synonym for the LD instruction (see [LD LDH LDW LDB LDD](#)).

Syntax Example

`POP %r1;` this instruction is equivalent to `LD.AB %r1, [%sp, 4]`.

POP_S

Function

Pop from Stack

Extension Group

BASELINE

Operation

$b = *SP; SP = SP + 4;$

Instruction Format

op b

Syntax Example

POP_S b

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a 32-bit word memory load from the address specified in the implicit Stack Pointer, SP (r28), and place the result into the destination register (b).

On completion, the implicit stack pointer is automatically incremented by 4-bytes ($SP = SP + 4$).

The status flags are not updated with this instruction.

This instruction is equivalent to a load word instruction LD when used with the following syntax:

LD.AB a, [SP,+4]

Pseudo Code

```
dest = Memory(SP, 4)    /* POP_S */
SP   = SP + 4
```

Assembly Code Example

```
POP_S r1      ; Load word from memory at address SP and write this word to
               ;r1, and then add 4 to SP
```

Syntax and Encoding

For more information about the encodings, see “Key for 32-bit Addressing Modes and Encoding Conventions” and “Key for 16-bit Addressing Modes and Encoding Conventions” .

Instruction Code		
POP_S	b	11000bbb11000001
POP_S	BLINK	11000RRR11010001

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 therefore 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.



Note

Software must not assume the PREALLOC instruction fills zero in 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 9-4](#) on page 696 and [Table 9-5](#) on page 697.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
PREALLOC<.aa>	[b,c]	00100bbb ¹ aa110RR10BBBCCCCC111110
PREALLOC<.aa>	[b,s9]	00010bbb ¹ ssssssssSBBB0aaRR111110

PREFETCH

Function

Prefetch from Memory

Extension Group

BASELINE

Operation

$\text{addr} = \text{b} + \text{c}; \text{temp} = * \text{addr};$

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 writeback 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 6-11 on page 267 lists the address writeback 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 9-4 PREFETCH Instruction Encodings

Class	<code><d></code>	<code><x></code>	Instruction
PREFETCH	0	0	PREFETCH
PREFETCH	0	1	PREALLOC
PREFETCH	1	0	PREFETCHW
PREFETCH	1	1	Reserved (currently deemed illegal)
Load	-	-	LD, LDB, LDH or LDD, depending on <code><zz></code>

The following combinations of load or prefetch-type modifier fields are illegal, and raise an Illegal Instruction exception.

- The `<x>` and `<d>` bits together select the prefetch-type opcode.
- The '-' character indicates a don't care case.
- The `<aa>` and `<zz>` fields operate identically for both prefetch and load instruction classes. Thus, prefetch-type instructions may have both data-size and address pre/post-increment or scaling, just as load instructions.

All illegal combinations of <d>, <x>, <aa>, and <zz> are shown in [Table 9-5](#). These combinations cover all load and prefetch-type instructions in 32-bit encodings.

Table 9-5 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions

Class	<d>	<x>	<aa>	<zz>	Instruction
PREFETCH	1	1	-	-	Reserved combination of PREFETCH opcode
Load	-	1	-	00	Cannot sign-extend from 32 bits
Load	-	1	-	11	Cannot sign-extend from 64bits
-	-	-	11	01	Cannot scale an address for byte data

Pseudo Code

```

if AA==0 then address = src1 + src2    /* PREFETCH */
if AA==1 then address = src1 + src2
if AA==2 then address = src1
if AA==3 then
    address = src1 + (src2 << 2)
if AA==1 or AA==2 then
    src1 = src1 + src2
DEBUG[LD] = 1

if NoFurtherLoadsPending() then      /* On Returning Load */
    DEBUG[LD] = 0

```

Assembly Code Example

```

PREFETCH [r1,4]    ; Prefetch the cache line from memory containing
                    address r1+4

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

PREFETCH<.aa>	[b, c]	00100 bbb aa110RR00 BBB CCCCC111110
PREFETCH<.aa>	[b, s9]	00010 bbb ssssssss S

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 therefore 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]    ; Prefetch the cache line from memory containing
                    address r1+4 with an intention to write
```

Syntax and Encoding

See [Table 9-4 on page 696](#) and [Table 9-5 on page 697](#). For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
PREFETCHW<.aa>	[b, c]	00100bbb ^{aa} 110RR01BBBCCCCC111110
PREFETCHW<.aa>	[b, s9]	00010bbb ^{ssssssss} SBBB1 ^{aa} RR0111110

PUSH

Function

Synonym for [ST 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
PUSH_S	b	11000bbb11100001
PUSH_S	blink	11000RRR11110001

QMACH

Function

Quad 16x16 multiply and accumulate. For the DSP version, see [QMACH](#).

Extension Group

(MPY_OPTION == 9) or (dsp_wide==true)

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"](#).

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 the 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 (r58 and r59).

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 (cc)== true then                                     /*QMACH*/
    result = acc + (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3)
    A = result
    acc = result
```

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



Note

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .
For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
QMACH<.f>	a,b,c	00101bbb00110100FBBBCCCCCAAAAA
QMACH<.f>	a,b,u6	00101bbb01110100FBBBuuuuuuAAAAAA
QMACH<.f>	b,b,s12	00101bbb10110100FBBBsssssssSSSSSS
QMACH<.cc><.f>	b,b,c	00101bbb11110100FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

QMACHU

Function

Quad unsigned 16x16 multiply and accumulate. For the DSP version, see [QMACHU](#).

Extension Group

(MPY_OPTION == 9) or (dsp_wide==true)

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"](#).

Instruction Format

op A,B,C

Syntax Example

QMACHU <.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

The B and C operands specify 64-bit values, each containing four 16-bit elements. Each pair of four elements in B and C is 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 (r58 and r59).

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)==true then                                     /*QMACHU*/
    result = acc + (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3)
    A = result
    acc = result
```

Assembly Code Example

```
QMACHU r0,r2,r4    ; quad 16x16 unsigned multiplication and
                   ;accumulation of (r3,r2) and (r5,r4). The result is
                   ;stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
QMACHU<.f>	a,b,c	00101bbb00110101FBBBCCCCCAAAAAA
QMACHU<.f>	a,b,u6	00101bbb01110101FBBBuuuuuuAAAAAA
QMACHU<.f>	b,b,s12	00101bbb10110101FBBBssssssSSSSSS
QMACHU<.cc><.f>	b,b,c	00101bbb11110101FBBBCCCCC0QQQQQ
QMACHU<.cc><.f>	b,b,u6	00101bbb11110101FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

QMPYH

Function

Quad 16x16 multiplication. For the DSP version, see [QMPYH](#).

Extension Group

(MPY_OPTION == 9) or (dsp_wide==true)

Operation

```
if (cc) {
    result = (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"](#).

Instruction Format

op A,B,C

Syntax Example

QMPYH <.f> A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	= Cleared

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 then summed and assigned to the destination register pair and the 64-bit accumulator (r58 and r59).

Flags are updated only if the set flags suffix (.F) is used. The sign (N) flag is set to the resulting sign of the accumulator. This instruction always clears the V flag.

Pseudo Code

```
if(cc) {
    result = (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3);
    A = result;
    acc = result;
}
```

/* QMPYH*/

Assembly Code Example

```
QMPYH r0,r2,r4
; quad 16x16 multiplication of
; (r3,r2) and (r5,r4). The result
; is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .
For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
QMPYH<.f>	a,b,c	00101bbb00110000FBBBCCCCCAAAAAA
QMPYH<.f>	a,b,u6	00101bbb01110000FBBBuuuuuuAAAAAA
QMPYH<.f>	b,b,s12	00101bbb10110000FBBBssssssSSSSSS
QMPYH<.cc><.f>	b,b,c	00101bbb11110000FBBBCCCCC0QQQQQ
QMPYH<.cc><.f>	b,b,u6	00101bbb11110000FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#)

QMPYHU

Function

Quad unsigned 16x16 multiply. For the DSP version, see [QMPYHU](#).

Extension Group

(MPY_OPTION == 9) or (dsp_wide==true)

Operation

```
if (cc) {
    result = (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"](#).

Instruction Format

op A,B,C

Syntax Example

QMPYHU <.f> A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Cleared

Description

The B and C operands specify 64-bit values, each containing four unsigned 16-bit elements. Each pair of four elements in B and C is multiplied to form four 32-bit unsigned products. The four products are then summed and assigned to the destination register pair and the 64-bit accumulator (r58 and r59).

Flags are updated only if the set flags suffix (.F) is used. This instruction always clears the overflow (V) flag.

Pseudo Code

```
if(cc) {
    result = (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3);
    A = result;
    acc = result;
}
```

Assembly Code Example

```
QMPYHU r0,r2,r4    ; quad 16x16 unsigned multiplication of (r3,r2) and
                   ; (r5,r4). The results is stored in (r1,r0)
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code

QMPYHU<.f>	a,b,c	00101bbb00110001FBBBCCCCCAAAAAA
QMPYHU<.f>	a,b,u6	00101bbb01110001FBBBuuuuuuAAAAAA
QMPYHU<.f>	b,b,s12	00101bbb10110001FBBBssssssSSSSSS
QMPYHU<.cc><.f>	b,b,c	00101bbb11110001FBBBCCCCC0QQQQQ
QMPYHU<.cc><.f>	b,b,u6	00101bbb11110001FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

RCMP

Function

Reverse Comparison

Extension Group

BASELINE

Operation

if (cc) $c - b$;

Instruction Format

op b,c

Syntax Example

RCMP<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

A reverse comparison is performed by subtracting source operand 1 (B) from source operand 2 (C) and subsequently updating the flags.

There is no destination register. Therefore, the result of the subtract is discarded.



Note

RCMP always sets the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                                /* RCMP */
  alu = src2 - src1
  Z_flag = if alu==0 then 1 else 0
  N_flag = alu[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
RCMP r1,r2    ; Subtract r1 from r2 and set the flags on the result
```

Syntax and Encoding

RCMP always set the flags even though there is no associated flag setting suffix.

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
RCMP	b, c	00100bbb0000011011BBBCCCCCRRRRRR
RCMP	b, u6	00100bbb0100011011BBBuuuuuuRRRRRR
RCMP	b, s12	00100bbb1000011011BBBssssssSSSSSS
RCMP<.cc>	b, c	00100bbb1100011011BBBCCCCC0QQQQQ
RCMP<.cc>	b, u6	00100bbb1100011011BBBuuuuuu1QQQQQ

REM

Function

2's complement integer Remainder. For the DSP version, see [REM](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

if (cc) then $a = b \% c$;

Instruction Format

op a, b, c

Syntax Example

REM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if divisor is zero or if an arithmetic overflow occurs

Description

If the divisor (c) is non-zero, the destination register is assigned the remainder obtained by signed integer division of source operand (b) by source operand (c).

If the divisor (c) is zero, the destination register is never updated.

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

An arithmetic overflow is signaled if the quotient of (b / c) cannot be represented in 32 bits. This overflow only occurs in one specific case, which is the division of the largest representable negative value (0x80000000) by -1. If an arithmetic overflow occurs, the overflow flag (V) is set to 1.

If a division-by-zero is attempted, or if the result cannot be represented in 32 bits, when the EV_DivZero exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, “Implementation Behavior on Division Overflow”](#).

Pseudo Code

```

if (cc == true) {
    if ((src2 != 0) && ((src1 != 0x80000000) || (src2 != 0xffffffff)))
    {
        dest = src1 % src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = dest[31];
            V = 0;
        }
    } else {
        if ((src2 == 0) && (STATUS32.DZ == 1))
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1){
                V = 1;
            }
        }
    }
}
}

```

Assembly Code Example

```
REM r1,r2,r3                ; r1 is assigned r2 % r3
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

REM<.f>	a,b,c	00101 bbb 00 001000 F BBB CCCCC AAAAAA
REM<.f>	a,b,u6	00101 bbb 01 001000 F BBB uuuuuu AAAAAA
REM<.f>	b,b,s12	00101 bbb 10 001000 F BBB ssssss SSSSSS
REM<.cc><.f>	b,b,c	00101 bbb 11 001000 F BBB CCCCC 0QQQQQ
REM<.cc><.f>	b,b,u6	00101 bbb 11 001000 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#)

REMU

Function

Unsigned integer Remainder. For the DSP version, see [REMU](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

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, see [Appendix Table B-1, "Implementation Behavior on Division Overflow"](#).

If the flag-update bit is set, an unsigned REMU operation always clears the N-flag.

Pseudo Code

```
if (cc == true) {
    if (src2 != 0) {
        dest = src1 % src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = 0;
            V = 0;
        }
    } else {
        if (STATUS32.DZ == 1)
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1){
                V = 1;
            }
        }
    }
}
```

Assembly Code Example

```
REMU r1,r2,r3                ; r1 is assigned r2 % r3
```

Syntax and Encoding

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
REMU<.f>	a,b,c	00101bbb00001001FBBBCCCCCAAAAAA
REMU<.f>	a,b,u6	00101bbb01001001FBBBuuuuuuAAAAAA
REMU<.f>	b,b,s12	00101bbb10001001FBBBssssssSSSSSS
REMU<.cc><.f>	b,b,c	00101bbb11001001FBBBCCCCC0QQQQQ
REMU<.cc><.f>	b,b,u6	00101bbb11001001FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

RLC

Function

Rotate Left Through Carry

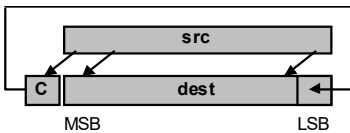
Extension Group

BASELINE

Operation

$b = c \ll 1; b = b \mid C; C = c \gg 31;$

Note: In this instruction, C represents the carry bit.



Instruction Format

op b,c

Syntax Example

RLC<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Undefined

Description

Rotate the source operand (c) left by one and place the result in the destination register (b). The carry flag is shifted into the least-significant bit of the result, and the most-significant bit of the source is placed in the carry flag. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

dest = src << 1                                /* RLC */
dest[0] = C_flag
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[31]
  V_flag = UNDEFINED

```

Assembly Code Example

```

RLC r1,r2    ; Rotate left through carry contents of r2 by one bit and
              ;write result into r1

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

RLC<.f>	b, c	00100bbb00101111FBBBCCCCC001011
RLC<.f>	b, u6	00100bbb01101111FBBBuuuuuu001011

ROL

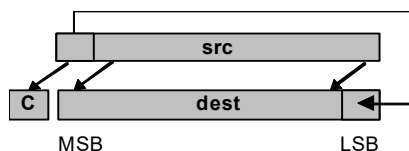
Function

Rotate Left

Extension Group

BASELINE

Operation

$$b = (c \ll 1) \mid (c \gg 31);$$


Instruction Format

op b,c

Syntax Example

ROL<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set to most-significant bit of source operand
V		= Unchanged

Description

Rotate the source operand (c) 1 bit to the left and place the result in the destination register (b).

The most-significant bit of the source operand is copied to the carry flag.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (src << 1) | (src >> 31)          /* ROL */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[31]
```

Assembly Code Example

```
ROL r1,r2    ; Rotate r2 1 place to the left, writing the result to r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

ROL<.f>	b,c	00100	bbb00	101111	FBBB	CCCCC	001101
ROL<.f>	b,u6	00100	bbb01	101111	FBBB	uuuuuu	001101

ROL8

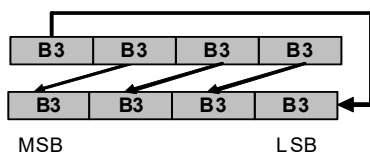
Function

Rotate Left 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$$b = (c \ll 8) \mid (c \gg 24);$$


Instruction Format

op b,c

Syntax Example

ROL8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Rotate the source operand 8 places to the left.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (src << 8) | (src >> 24)          /* ROR8 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
ROL8 r1,r2    ; Rotate r2 8 places to the left, placing
               ; result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

ROL8<.f>	b,c	00101	bbb00	101111	FBBB	CCCCC	010000
ROL8<.f>	b,u6	00101	bbb01	101111	FBBB	uuuuuu	010000

ROR

Function

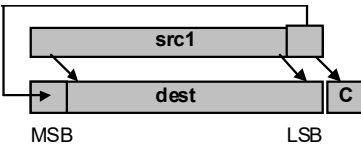
Rotate Right

Extension Group

BASELINE

Operation

$C = c \ \& \ 1; \ b = c \gg 1 \mid c \ll 31;$



Instruction Format

op b,c

Syntax Example

ROR<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Rotate the source operand (c) right by one and place the result in the destination register (b).
The least-significant bit of the source operand is copied to the carry flag.
Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 1                                /* ROR */
dest[31] = src[0]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[0]
```

Assembly Code Example

```
ROR r1,r2    ; Rotate right contents of r2 by one bit and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ROR<.f>	b,c	00100bbb00101111FBBBCCCCC000011
ROR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000011

ROR Multiple

Function

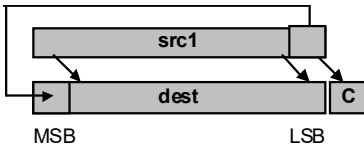
Multiple Rotate Right

Extension Group

SHIFT_OPTION 2 or 3

Operation

if (cc) {C = b >> (c-1) & 1; a = b >> c | b << (31-c);}



Instruction Format

op a, b, c

Syntax Example

ROR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Rotate operand (b) right by (c) places and place the result in the destination register a. Only the bottom 5 bits of (c) are used as the shift value.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc=true then                                     /* ROR */
  dest = src1 >> (src2 & 31)                       /* Multiple */
  dest [31:(31-src2)] = src1 [(src2-1):0]
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = if src2==0 then 0 else src1[src2-1]

```

Assembly Code Example

```
ROR r1,r2,r3 ;Rotate right contents of r2 by r3 bits and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

ROR<.f>	a,b,c	00101bbb0000011FBBBCCCCC	AAAAAA
ROR<.f>	a,b,u6	00101bbb01000011FBBB	uuuuuuAAAAAA
ROR<.f>	b,b,s12	00101bbb10000011FBBB	ssssssSSSSSS
ROR<.cc><.f>	b,b,c	00101bbb11000011FBBBCCCCC	0QQQQQ
ROR<.cc><.f>	b,b,u6	00101bbb11000011FBBB	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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ROR8

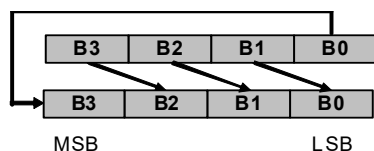
Function

Rotate right 8

Extension Group

SHIFT_OPTION 1 or 3

Operation

$$b = (c \gg 8) \mid (c \ll 24);$$


Instruction Format

op b,c

Syntax Example

ROR8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Rotate the source operand 8 places to the right.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (src >> 8) | (src << 24)          /* ROR8 */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
ROR8 r1,r2    ; Rotate r2 8 places to the right, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

```
ROR8<.f>  b,c      00101bbb00101111FBBBCCCCC010001
```

```
ROR8<.f>  b,u6     00101bbb01101111FBBBuuuuuu010001
```

RRC

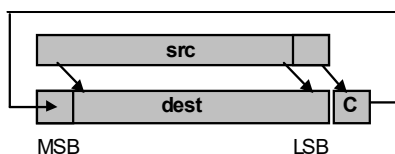
Function

Rotate Right through Carry

Extension Group

BASELINE

Operation

$$b = c \gg 1 \mid C \ll 31; C = c \& 1;$$


Instruction Format

op b,c

Syntax Example

RRC<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Rotate the source operand (c) right by one and place the result in the destination register (a).

The carry flag is shifted into the most-significant bit of the result, and the least-significant bit of the source is placed in the carry flag.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 1                                /* RRC */
dest[31] = C_flag
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = src[0]
```

Assembly Code Example

```
RRC r1,r2    ; Rotate right through carry contents of r2 by one bit and
              ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

RRC<.f>	b,c	00100bbb00101111FBBBCCCCC000100
RRC<.f>	b,u6	00100bbb01101111FBBBuuuuuu000100

RSUB

Function

Reverse Subtract

Extension Group

BASELINE

Operation

if (cc) a = c - b;

Instruction Format

op a, b, c

Syntax Example

RSUB<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Subtract source operand 1 (b) from source operand 2 (c) and place the result in the destination register (a).

If the carry flag is set upon performing the subtract, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* RSUB */
    dest = src2 - src1
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
RSUB r1,r2,r3    ; Subtract contents of r2 from r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

RSUB<.f>	a, b, c	00100bbb00001110FBBBCCCCCAAAAA
RSUB<.f>	a, b, u6	00100bbb01001110FBBBuuuuuuAAAAA
RSUB<.f>	b, b, s12	00100bbb10001110FBBBssssssSSSSSS
RSUB<.cc><.f>	b, b, c	00100bbb11001110FBBBCCCCC0QQQQQ
RSUB<.cc><.f>	b, b, u6	00100bbb11001110FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

RTIE

Function

Return from Interrupt/Exception

Extension Group

BASELINE

Operation

```
if ((HAS_INTERRUPTS == 0) || (STATUS32.AE) || (AUX_IRQ_ACT[15:0] == 0))
    ReturnFromException()
else
    if ((AUX_IRQ_ACT[0]) && (FIQ_OPTION))
        if (IRQ_PRIORITY_PENDING[0])
            ServiceAnotherFastInterrupt()
        else
            ReturnFromFastInterrupt()
    else
        if (PendingNewIntrPreemptsReturnLevel)
            ServiceAnotherInterrupt()
        else
            ReturnFromInterrupt()
```



Note

For information about the conditions and function used in this section, see the sections: [Fast Interrupt Exit](#), [Returning from Regular Interrupts](#), [Exception Exit](#), and [Exceptions and Delay Slots](#).

Instruction Format

Zero operands

Syntax Example

RTIE

STATUS32 Flags Affected

IE		• = Set according to status register update
AD		• = Set according to status register update
RB[2:0]		• = Set according to status register update
ES		• = Set according to status register update
SC		• = Set according to status register update
DZ		• = Set according to status register update
L		• = Set according to status register update

Z	•	= Set according to status register update
N	•	= Set according to status register update
C	•	= Set according to status register update
V	•	= Set according to status register update
U	•	= Set according to status register update
DE	•	= Set according to status register update
AE	•	= Set according to status register update
E[3:0]	•	= Set according to status register update
H	•	= 0

Description

The return from interrupt or exception instruction, RTIE, allows exit from interrupt and exception handlers, and also allows the processor to switch from kernel mode to user mode.

The RTIE instruction is available only in kernel mode. Using this instruction in the user mode raises a [Privilege Violation, Kernel Only Access](#) exception.

The interrupt and exception handlers use the RTIE to exit an interrupt or exception. The RTIE instruction restores previous context including the program counter, status register and, optionally, selected core registers depending on whether the return is from an exception, a fast or regular interrupt, and to what machine operating level the processor is returning. If an RTIE is executed on a processor which has no interrupts configured, the RTIE performs a return from exception as illustrated in the pseudo-code function, `ReturnFromException()`.

Bits in the STATUS32, AUX_IRQ_ACT and IRQ_PRIORITY_PENDING registers are provided to allow the RTIE instruction to determine what pre-interrupt or exception machine state to restore and from where to reload the state. Machine state is restored from ERET, ERSTATUS, and ERBTA registers when returning from an exception, from ILINK, STATUS32_P0 registers when returning from a fast interrupt and, from a user stack or kernel stack when returning from a regular interrupt. When returning from a regular interrupt, selected core and auxiliary registers may also be restored from the stack depending on bits in the AUX_IRQ_CTRL register.

When the core is executing a microcoded epilog sequence of an RTIE, accesses the stack in memory, it is possible that an exception such as a memory protection violation or stack checking violation may occur. So, the epilog sequence is designed to be re-startable such that the stack pointer is not modified until an epilog sequence is successfully completed.

On return from a fast interrupt the processor restores the STATUS32 register including the RB field. When more than one bank of core registers are configured, the processor implicitly switches to the register bank defined by the RB field in the restored STATUS32 register. A detailed discussion of the actions taken on interrupt and exception entry and exit is available in sections:

- [Fast Interrupt Entry](#)
- [Fast Interrupt Exit](#)

- ❑ [Regular Interrupt Entry](#)
- ❑ [Returning from Regular Interrupts](#)
- ❑ [Exception Entry](#)
- ❑ [Exception Exit](#)
- ❑ [Exceptions and Delay Slots](#)

As exceptions are permitted between a branch/jump and an executed delay slot instruction, special branch target address registers are used for exception handler returns.

If the STATUS32[DE] bit is set as a result of the RTIE instruction, the processor is put back into a state where a branch with a delay slot is pending. The target of the branch is contained in the BTA register which is restored from the appropriate Exception Return BTA register (ERBTA).

Pseudo Code Example

To understand the pseudo code, refer to the following notation:

- ❑ P is the processor's current operating priority defined by the index of least-significant bit set (with value of 1) in AUX_IRQ_ACT.ACTIVE. P is 16 if no interrupts are active.
- ❑ Q is the index of the least-significant bit set (with a value of 1) in AUX_IRQ_ACT.ACTIVE when bit P is cleared. Q is 16 if P is the only bit set to 1 in AUX_IRQ_ACT.ACTIVE.
- ❑ PI is the highest priority pending and enabled interrupt, and its priority is W.
- ❑ If the AE bit is set in STATUS32, or if there is no active interrupt handler, an RTIE instruction assumes a return from exception.
- ❑ If the AE bit is not set, and if there is an active interrupt 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)
    else
        ReturnFromInterrupt (PI, W)

ReturnFromException()
Next_PC = ERET
STATUS32 = ERSTATUS
BTA = ERBTA
if (ERSTATUS.U)                                //check for User mode
    AEX R28, [AUX_USER_SP]                      //restore User SP
Jump (Next_PC)
```

Assembly Code Example

```
RTIE                                ; Return from interrupt/exception
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
RTIE    00100100011011110000RRRRRR111111
```

SBC

Function

Subtract with Carry

Extension Group

BASELINE

Operation

if (cc) $a = (b - c) - C$;

Instruction Format

op a, b, c

Syntax Example

SBC<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Subtract source operand 2 (c) from source operand 1 (b) and also subtract the state of the carry flag C. If C is set, subtract 1; otherwise, subtract 0. Place the result in the destination register a.

If the carry flag is set upon performing the subtract, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* SBC */
    dest = (src1 - src2) - C_flag
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```


Assembly Code Example

```
SBC r1,r2,r3    ; Subtract with carry contents of r3 from r2 and write
                ;result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SBC<.f>	a,b,c	00100bbb00000011FBBBCCCCCAAAAAA
SBC<.f>	a,b,u6	00100bbb01000011FBBBuuuuuuAAAAAA
SBC<.f>	b,b,s12	00100bbb10000011FBBBssssssSSSSSS
SBC<.cc><.f>	b,b,c	00100bbb11000011FBBBCCCCC0QQQQQ
SBC<.cc><.f>	b,b,u6	00100bbb11000011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

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. 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.

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 generates exclusive transactions (in AXI this means the command attribute AxLOCK is asserted). For more information, see [Atomic Memory Operations](#) section.

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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

```
SCOND<.di>  b, [c]    00100bbb00101111DBBBCCCCC010001
SCOND<.di>  b, [u6]   00100bbb01101111DBBBuuuuuu010001
```

SCONDD

Function

Store conditional on 64-bit data

Extension Group

(ATOMIC_OPTION == 1) and (LL64_OPTION==1)

Operation

`_scondd (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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

SCONDD<.di>	b, [c]	00100	bbb00	101111	D	BBB	CCCCCC	010011
SCONDD<.di>	b, [u6]	00100	bbb01	101111	D	BBB	uuuuuu	010011

SETcc

Function

Compute Boolean relation between operands

Extension Group

CODE_DENSITY

Operation

$\text{dest} \leftarrow \text{Relation}(\text{src1}, \text{src2})$

Instruction Format

op a, b, c

Syntax Example

SETcc<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if both source operands are equal
N	•	= Set if (src1 – src2) is negative
C	•	= Set if carry is generated by (src1 – src2)
V	•	= Set if an overflow is generated by (src1 – src2)

Description

Compare the two signed source operands (b) and (c) using the selected relational operator, and place the Boolean result (0 or 1) in the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* SETcc */
dest = (src1 Relop src2)
  if F==1 then
    Z_flag = if (src1 == src2) then 1 else 0
    N_flag = if (src1 < src2) then 1 else 0
    V_flag = Overflow (src1-src2)
    C_flag = CarryOut (src1-src2)

```

Assembly Code Example

```
SETLT r1,r2,r3    ; Set r1 to 1 if r2 < r3 otherwise set r1 to 0
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SETcc<.f>	a,b,c	00100bbb00iiiiiiiFBBBCCCCCAAAAAA
SETcc<.f>	a,b,u6	00100bbb01iiiiiiiFBBBuuuuuuAAAAAA
SETcc<.f>	b,b,s12	00100bbb10iiiiiiiFBBBssssssSSSSSS
SETcc<.cc><.f>	b,b,c	00100bbb11iiiiiiiFBBBCCCCC0QQQQQ
SETcc<.cc><.f>	b,b,u6	00100bbb11iiiiiiiFBBBuuuuuu1QQQQQ

SETcc encodings

Sub-opcode	iiii	Instruction mnemonic	Description	Operand type
0x38	111000	SETEQ	set if equal	Signed or unsigned
0x39	111001	SETNE	set if not equal	Signed or unsigned
0x3a	111010	SETLT	set if less than	Signed
0x3b	111011	SETGE	set if greater or equal	Signed
0x3c	111100	SETLO	set if lower than	Unsigned
0x3d	111101	SETHS	set if higher or same	Unsigned
0x3e	111110	SETLE	set if less than or equal	Signed
0x3f	111111	SETGT	set if greater than	Signed

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SETI

Function

SETI enables interrupts, and if a source operand is mentioned, sets the interrupt threshold in the STATUS32 register.

Extension Group

HAS_INTERRUPTS == 1

Operation

```
{
  if (c[5] == 1)
    STATUS32.E ← [3:0]
    STATUS32.IE ← c[4]
  else
    STATUS32.IE ← 1
    if (c[4] == 1)
      STATUS32.E ← c[3:0]
}
```

Instruction Format

op c

Syntax Example

SETI c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

SETI takes a single u6 or C source register operand that indicates whether the interrupt priority level is set when enabling interrupts, and if so to what level.

When an ARCV2-based processor is configured without interrupts (HAS_INTERRUPTS = 0), executing a SETI instruction raises an [Illegal Instruction](#) exception.

The SETI instruction is available only in kernel mode. Using this instruction in the user mode raises a [Privilege Violation, Kernel Only Access](#) exception.

□

Pseudo Code

```
{
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]
}
```

/* SETI
*/

Assembly Code Example

```
SETI R0      ;enable interrupts and restore the preempting interrupt
              ;priority value.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
SETI	c	00100110001011110000CCCCC111111
SETI	u6	00100110011011110000uuuuuu111111

SETI takes a 6-bit operand. Bit 5 indicates whether interrupt enables are being restored (1) or forced to specific values (0).

When forcing the interrupt enable to 1, the STATUS32.E field can be optionally set according to bits [3:0] of the SETI operand.

Bit5	Bit4	Purpose
0	0	Enable interrupts without changing the enabled level
0	1	Enable interrupts and also set the enabled level
1	x	Restore STATUS32.IE and STATUS32.E using bits 4 and [3:0] respectively

The assembler must encode a SETI without operands as SETI u6, with u6 == 0 to provide a concise way of specifying “enable interrupts”.

SETI is a serializing instruction similar to the KFLAG instruction. This behavior means after this instruction commits, the pipeline is flushed.

SEXB

Function

Sign Extend Byte

Extension Group

BASELINE

Operation

$$b = (c \ll 24) \gg 24;$$

Instruction Format

op b,c

Syntax Example

SEXB<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Sign extend the byte contained in the least-significant 8 bits of source operand (c) to a full 32-bit word value and place the result into the destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest[7:0] = src[7:0]           /* SEXB */
dest[31:8] = src[7]
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
SEXB r1,r2    ; Sign extend the bottom 8 bits of r2 and write result to r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SEXB<.f> b,c	00100	bbb00101111FBBBCCCCC000101
SEXB<.f> b,u6	00100	bbb01101111FBBBuuuuuu000101
SEXB_S b,c	01111	bbbccc01101

SEXH, SEXW

Function

Sign Extend Half-word 16-bits to word



Note

The SEXW mnemonic is deprecated.

Extension Group

BASELINE

Operation

$$b = (c \ll 16) \gg 16;$$

Instruction Format

op b,c

Syntax Example

SEXH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Sign extend the 16-bit half-word contained in the least-significant 16 bits of source operand (c) to a full 32-bit word value and place the result into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest[15:0] = src[15:0]           /* SEXH */
dest[31:16] = src[15]
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
SEXH r1,r2    ; Sign extend the bottom 16 bits of r2 and write result to r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SEXH<.f> b,c		00100bbb00101111FBBBCCCCC000110
SEXH<.f> b,u6		00100bbb01101111FBBBuuuuuu000110
SEXH_S b,c		01111bbbccc01110

SLEEP

Function

Enter Sleep State

Extension Group

BASELINE

Operation

DEBUG[ZZ] = 1;

Instruction Format

op c

Syntax Example

SLEEP

STATUS32 Flags Affected

IE	• Updated depending on SLEEP operand as explained in Table 9-6
E[3:0]	• Updated depending on SLEEP operand as explained in Table 9-6
Z	= Unchanged
N	= Unchanged
C	= Unchanged
V	= Unchanged
ZZ	• = 1
SM	• Updated depending on SLEEP operand as explained in Table 9-6

Description

The ARCV2-based processor enters the sleep state when the processor encounters the SLEEP instruction while operating in a kernel mode. The processor stays in the sleep state until an interrupt or restart occurs. Power consumption may be reduced during sleep state because the processor is inactive, and the RAMs may be disabled. The SLEEP instruction is available only in kernel mode. Using this instruction in the user mode raises a [Privilege Violation, Kernel Only Access](#) exception.

SLEEP instruction operands

The SLEEP instruction is a single operand instruction without flags. A SLEEP instruction without a source operand is encoded as SLEEP 0.

The SLEEP instruction is serializing, which means the SLEEP instruction completes and then flushes the pipeline.

In sleep state, the sleep state flag (ZZ) is set, and the host interface operates in the normal way, allowing access to the DEBUG and the STATUS registers. The host interface also has the ability to halt the processor. The host cannot clear the sleep state flag, but the host can wake the processor by halting the processor, and then restarting it. The program counter (PC) points to the next instruction in sequence after the sleep instruction.

The processor wakes up from sleep state when an interrupt occurs, a non-maskable imprecise exception is triggered, or when the processor is restarted. If an interrupt wakes the processor, the ZZ flag is cleared, the interrupt routine is serviced, and execution resumes at the instruction in sequence after the SLEEP instruction. When a processor is started after being halted, the ZZ flag is cleared.

The SLEEP instruction is a NOP during the single-step mode. Consequently, the behavior is of NOP instruction and the IE, E[3:0], ZZ & SM bits are unchanged.

Every single-step operation is a restart and the ARCV2-based processor wakes up at the next single-step.

Three operand bits ([7:5]) of the SLEEP instruction indicate to the system the specific sleep mode through the sys_sleep_mode_r[2:0] output vector.

Figure 9-2 SLEEP Operand

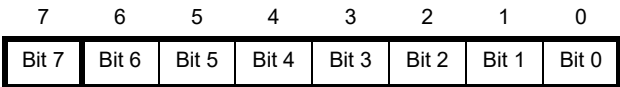


Table 9-6 SLEEP Operand

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()
```

Assembly Code Example

The SLEEP instruction can be placed anywhere in the instruction sequence.

Example 9-5 Sleep placement in code

```
SUB r2, r2, 0x1
ADD r1, r1, 0x2
SLEEP
...
```

[Example 9-6](#) illustrates the use of the SLEEP instruction with each combination of interrupts levels enabled or disabled during sleep.

Example 9-6 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 ;prioriry level to 4. Any interrupt with a priority ;equal to or higher than 4 awakens the ;processor.

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
SLEEP	c	00100001001011110000CCCCC111111
SLEEP	<u6>	00100001011011110000uuuuuu111111

SR

Function

Store to Auxiliary Register

Extension Group

BASELINE

Operation

`_sr(b,c);`

Instruction Format

`op b,c`

Syntax Example

`SR b,[c]`

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Store the 32-bit Word held in source operand 1 (b) into the auxiliary register whose address is obtained from the source operand 2 (c).

Pseudo Code

```
Aux_reg(src2) = src1    /* SR */
```

Assembly Code Example

```
SR r1,[r2]    ; Store contents of r1 into auxiliary register pointed by r2
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SR	b, [c]	00100bbb00101011RBBBCCCCCRRRRRR
SR	b, [u6]	00100bbb01101011RBBBuuuuuuRRRRRR
SR	b, [s12]	00100bbb10101011RBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

ST ST STH STB 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 6-14](#) on page [268](#) lists the data size modes.



Note

For more information about the rules governing non-aligned memory references, see [Data Layout in Memory](#).

If the processor contains a data cache, store requests can bypass the cache by using the <.di> syntax.

Table 6-13 on page 268 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 writeback mode can be selected by use of the <.aa> syntax.

Table 6-11 on page 267 lists the address writeback 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 or STL 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
ST<zz><.aa><.di>	c, [b]	00011bbb000000000BBBCCCCC0DaaZZ0
ST<zz><.aa><.di>	c, [b, s9]	00011bbbssssssssSBBBCCCCC0DaaZZ0
ST<zz><.aa><.di>	w6, [b]	00011bbb000000000BBBwwwwwDaaZZ1
ST<zz><.aa><.di>	w6, [b, s9]	00011bbbssssssssSBBBwwwwwDaaZZ1
ST_S	b, [SP, u7]	11000bbb010uuuuu
ST_S	c, [b, u7]	10100bbbcccuuuuu
STB_S	b, [SP, u7]	11000bbb011uuuuu



Note

The encoding of STB_S b, [SP, u7] includes a 5-bit constant, but this constant gets shifted left by two bit positions, as all stack addresses are expected to be four-byte-aligned. This alignment makes the constant a u7 when considering the range of offsets, although the lower two bits are always 0.

Encodings supported when any CODE DENSITY option is enabled

ST_S	R0, [GP, s11]	01010SSSSS10sss

Encodings supported when CODE DENSITY option is 2 or 3

STB_S	c, [b, u5]	10101bbbcccuuuuu
STH_S	c, [b, u6]	10110bbbcccuuuuu

Encodings supported by LL64_OPTION

Instruction Code		
STD<.aa><.di>	c, [b]	00011bbb000000000BBBCCCCC0Daa110
STD<.aa><.di>	c, [b, s9]	00011bbbssssssssSBBBCCCCC0Daa110
STD<.aa><.di>	w6, [b]	00011bbb000000000BBBwwwwwDaa111
STD<.aa><.di>	w6, [b, s9]	00011bbbssssssssSBBBwwwwwDaa111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SUB

Function

Subtract

Extension Group

BASELINE

Operation

if (cc) $a = b - c$

Instruction Format

op a, b, c

Syntax Example

SUB<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Subtract source operand 2 (c) from source operand 1 (b) and place the result in the destination register a.

SUB_S.NE is executed when the Z flag is equal to zero.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.



Note

For the 16-bit encoded instructions that work on the stack pointer (SP), the offset is aligned to 32-bit. For example, SUB_S sp, sp. u7 only needs to encode the top 5 bits because the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

Pseudo Code

```

if cc==true then                                /* SUB */
  dest = src1 - src2
  if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
    C_flag = Carry()
    V_flag = Overflow()

```

Assembly Code Example

```
SUB r1,r2,r3      ; Subtract contents of r3 from r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SUB<.f>	a,b,c	00100bbb0000010FBBBCCCCCAAAAAA
SUB<.f>	a,b,u6	00100bbb01000010FBBBuuuuuuAAAAAA
SUB<.f>	b,b,s12	00100bbb1000010FBBBssssssSSSSSS
SUB<.cc><.f>	b,b,c	00100bbb11000010FBBBCCCCC0QQQQQ
SUB<.cc><.f>	b,b,u6	00100bbb11000010FBBBuuuuuu1QQQQQ
SUB_S	c,b,u3	01101bbbccc01uuu
SUB_S.NE	b,b,b	01111bbb1100000
SUB_S	b,b,c	01111bbbccc00010
SUB_S	b,b,u5	10111bbb011uuuuu
SUB_S	SP,SP,u7	11000001101uuuuu

Encodings supported by the CODE DENSITY options

```
SUB_S          a,b,c          01001bbbccc10aaa
```

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SUB1

Function

Subtract with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b - (c \ll 1)$

Instruction Format

op a, b, c

Syntax Example

SUB1<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left shifted by 1) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                     /* SUB1 */
    dest = src1 - (src2 << 1)
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
SUB1 r1,r2,r3    ; Subtract contents of r3 left-shifted one bit from r2
                  ;and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SUB1<.f>	a,b,c	00100bbb00010111FBBBCCCCCAAAAAA
SUB1<.f>	a,b,u6	00100bbb01010111FBBBuuuuuuAAAAAA
SUB1<.f>	b,b,s12	00100bbb10010111FBBBssssssSSSSSS
SUB1<.cc><.f>	b,b,c	00100bbb11010111FBBBCCCCC0QQQQQ
SUB1<.cc><.f>	b,b,u6	00100bbb11010111FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SUB2

Function

Subtract with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b - (c \ll 2)$

Instruction Format

op a, b, c

Syntax Example

SUB2<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left-shifted by 2) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* SUB2 */
    dest = src1 - (src2 << 2)
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
SUB2 r1,r2,r3      ; Subtract contents of r3 leftshifted two
                   ; bits from r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SUB2<.f>	a,b,c	00100bbb00011000FBBBCCCCCAAAAAA
SUB2<.f>	a,b,u6	00100bbb01011000FBBBuuuuuuAAAAAA
SUB2<.f>	b,b,s12	00100bbb10011000FBBBssssssSSSSSS
SUB2<.cc><.f>	b,b,c	00100bbb11011000FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SUB3

Function

Subtract with Scaled Source

Extension Group

BASELINE

Operation

if (cc) $a = b - (c \ll 3)$

Instruction Format

op a, b, c

Syntax Example

SUB3<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left shifted by 3) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* SUB3 */
    dest = src1 - (src2 << 3)
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]
        C_flag = Carry()
        V_flag = Overflow()

```

Assembly Code Example

```
SUB3 r1,r2,r3    ; Subtract contents of r3 left shifted three
                  ; bits from r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
SUB3<.f>	a,b,c	00100bbb00011001FBBBCCCCCAAAAAA
SUB3<.f>	a,b,u6	00100bbb01011001FBBBuuuuuuAAAAAA
SUB3<.f>	b,b,s12	00100bbb10011001FBBBssssssSSSSSS
SUB3<.cc><.f>	b,b,c	00100bbb11011001FBBBCCCCC0QQQQQ
SUB3<.cc><.f>	b,b,u6	00100bbb11011001FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

SWAP

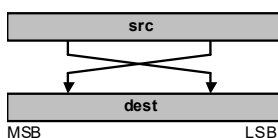
Function

Swap 16-bits half-words

Extension Group

SWAP_OPTION

Operation

$$b = c >> 16 \mid (c \& 0xFFFF) << 16;$$


Instruction Format

op b,c

Syntax Example

SWAP<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Swap the lower 16 bits of the operand with the upper 16 bits of the operand and place the result of that swap in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = SWAP(src)                /* SWAP */
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
SWAP r1,r2    ; Swap top and bottom 16 bits of r2 write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

SWAP<.f>	b,c	00101	bbb00	101111	FBBB	CCCCC	000000
SWAP<.f>	b,u6	00101	bbb01	101111	FBBB	uuuuuu	000000

SWAPE

Function

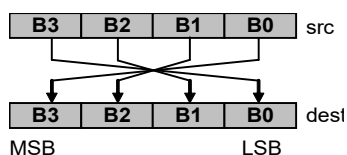
Swap byte ordering

Extension Group

SWAP_OPTION

Operation

b = swap order of bytes from c



Instruction Format

op b,c

Syntax Example

SWAPE<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Copy the source operand to the destination operand, with reversed byte ordering, or Endianness.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (ROL8(src) & 0x00ff00ff)          /* SWAPE */
      | (ROR8(src) & 0xff00ff00)
if F==1 then
    Z_flag = if dest==0 then 1 else 0
    N_flag = dest[31]
```

Assembly Code Example

```
SWAPE r1,r2    ; Take the bytes from r2, in reverse order and copy to r1.
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

SWAPE<.f>	b, c	00101	bbb00	101111	FBBB	CCCCC	001001
SWAPE<.f>	b, u6	00101	bbb01	101111	FBBB	uuuuuu	001001

SWI

Function

Software Interrupt or Software Breakpoint

Extension Group

SWI: BASELINE

SWI u6: OS_OPT_OPTION == 1. This option cannot be configured in the ARC EM processor, and therefore this instruction is decoded as an illegal instruction in the ARC EM processor.

Instruction Format

op

Syntax Example

SWI

SWI_S u6

STATUS32 Flags Affected

IE	•	= 0
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
E[3:0]		= Unchanged
U	•	= 0
AE	•	= 1

 **Note**

The SWI_S instruction affects the flags similar to SWI. The above flag status is also valid for SWI_S.

Operation

Software Breakpoint

Description

The SWI instruction provides a means to interrupt software execution at any point in the program. This instruction is typically used by a native debugger to insert soft breakpoints.

The only action performed by a SWI instruction is to raise an EV_SWI exception. The SWI instruction does not advance the program counter before the exception is raised. Therefore, the Exception Return Address register (see [Exception Return Address, ERET](#)) is set to the address of the SWI instruction itself. The Exception Fault Address register (see [Exception Fault Address, EFA](#)) is also set to point to the address of the SWI instruction.

The EV_SWI exception can be raised from user or kernel modes. When inserting a software breakpoint, the instruction at the appropriate address is replaced by a SWI instruction of the same size SWI_S for 16-bit instructions and SWI for 32-bit instructions. Before returning from the EV_SWI exception, the original instruction must replace the SWI or SWI_S instruction. When the exception handler is complete, program execution resumes at the address from which the SWI instruction was executed, and the original instruction executes as normal.

If a source is specified with the SWI_S instruction, the source operand is loaded into the exception cause register (see [Exception Cause Register, ECR](#)) as the cause parameter along with the cause code for a SWI instruction and the SWI vector number, otherwise the parameter field is zero.

The source value can be used to signal a type of command to any operating system that is running on the processor. Source values 0 to 62 can be used to encode distinct operating system calls.

Pseudo Code

```
ERET = currentPC                /* SWI, SWI_S */
ERSTATUS = STATUS32
if STATUS32[DE] == 1 then
    ERBTA = pending PC
ECR = 0x00 : 0x08 : 0x00 : src
EFA = PC
STATUS32[U] = 0
STATUS32[IE] = 0
STATUS32[AE] = 1
PC = INT_VECTOR_BASE + 0x20
```

Assembly Code Example

```
SWI      ; Software interrupt or breakpoint
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SWI 00100010011011110000RRRRRR111111

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.



Note

The SYNC instruction does not wait on memory operations started by other processors.

Use with Interrupts

The SYNC instruction can also be used to ensure that the interrupt request of a memory mapped peripheral has been cleared before an interrupt handler exits.

For example, consider the following scenario:

- ❑ A peripheral generates interrupt to the processor by setting a signal to true.
- ❑ The control registers for the peripheral are memory mapped

- ❑ The processor's interrupt unit is set to 'level sensitive' for this interrupt.
- ❑ The interrupt handler must clear the interrupt request signal before exiting

The SYNC instruction is used to ensure that the store to change the peripheral status happens before the interrupt exit

If the SYNC was not used, the peripheral may still be asserting the interrupt-request signal after the interrupt exit – therefore, an incorrect interrupt is generated.

To provide the instruction synchronize function, the instruction serializes on completion, meaning that the contents of the pipeline are discarded, and fetching restarted from the stored program counter value.

Use for Data Synchronization

For data synchronization, the purpose of the SYNC instruction is to ensure that all the following memory operations started by the processor have finished before any new operations (of any kind) can begin:

- ❑ All outstanding LD, ST and EX instructions
- ❑ All data cache operations, including line fills and flushes

Synchronizing Out-of-order State Updates

Any implementation that permits some instructions to update processor or extension state after the instruction has committed can be forced to wait for all such pending updates by use of the SYNC instruction.

For example, a floating-point extension may provide a multiply operation that commits 2 cycles after the operation is issued, but does not write its result until 3 cycles later. Another example is when a User Extension Instruction is pipelined over a period of 10 cycles, and directly modifies an extension register on completion. On completion of a SYNC instruction, the retirement of any such floating-point results, the update to extension registers, can be guaranteed to have completed for all instructions issued before the SYNC.

Pseudo Code

```
do                                                    /* SYNC */
  null
until not (load_pending or store_pending or dcache_fill or
           dcache_flush_instruction_retirement_pending)
```

Assembly Code Example

```
SYNC                                           ; Synchronize
```

Syntax and Encoding

The status flags are not updated with this instruction. Therefore, the flag setting field, F, is encoded as 0. For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

SYNC 00100011011011110000RRRRRR111111

TRAP_S

Function

Trap

Extension Group

BASELINE

Operation

Raise an exception

Instruction Format

op u6

Syntax Example

TRAP_S u6

STATUS32 Flags Affected

IE	•	= 0
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
E[3:0]		= Unchanged
U	•	= 0
AE	•	= 1

Description

The TRAP_S instruction raises an exception and calls any operating system procedures in kernel mode. Traps can be raised from user or kernel modes. The source operand is loaded into the exception cause register (see [Exception Cause Register, ECR](#)) as the cause parameter along with the cause code (0x0) for a trap and the trap vector number.

The source value can be used to signal a type of command to any operating system that is running on the processor. Source values 0 to 63 can be used to encode distinct operating system calls.

If implemented in the processor, the Exception Fault Address register (see [Exception Fault Address, EFA](#)) is set to point to the address of the trap instruction. The Exception Return Address register (see

[Exception Return Address, ERET](#)) is set to the address of the instruction immediately following the trap instruction.

When the exception handler has completed, program execution resumes at the instruction immediately following the trap instruction.

Pseudo Code

```
ERET = NEXTPC                                /* TRAP_S */
ERSTATUS = STATUS32
if STATUS32[DE] == 1 then
    ERBTA = pending PC
ECR = 0x00 : 0x09 : 0x00 : src
EFA = PC
STATUS32[U] = 0
STATUS32[IE] = 0
STATUS32[AE] = 1
PC = INT_VECTOR_BASE + 0x24
```

Assembly Code Example

```
TRAP_S 0 ; Trap
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

```
TRAP_S u6    01111uuuuuu11110
```

TST

Function

Test

Extension Group

BASELINE

Operation

if (cc) b & c;

Instruction Format

op b,c

Syntax Example

TST<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Bitwise AND of source operand 1 (b) with source operand 2 (c) and subsequently update the flags.

There is no destination register. Therefore, the result of the AND is discarded.



Note

TST and TST_S always set the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                                /* TST */
    alu = src1 AND src2
    Z_flag = if alu==0 then 1 else 0
    N_flag = alu[31]
```

Assembly Code Example

```
TST r1,r2    ; Logical AND r2 with r1 and set the flags on the result
```

Syntax and Encoding

The flag setting field, F, is always encoded as 1 for this instruction. For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
TST	b, c	00100bbb000010111BBBCCCCCRRRRRR
TST	b, u6	00100bbb010010111BBBuuuuuuRRRRRR
TST	b, s12	00100bbb100010111BBBssssssSSSSSS
TST<.cc>	b, c	00100bbb110010111BBBCCCCC0QQQQQ
TST<.cc>	b, u6	00100bbb110010111BBBuuuuuu1QQQQQ
TST_S	b, c	01111bbbccc01011

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

UNIMP_S

Function

Unimplemented Instruction

Extension Group

BASELINE

Operation

Raise an [Illegal Instruction](#) exception

Instruction Format

op

Syntax Example

UNIMP_S

STATUS32 Flags Affected

IE	•	= 0
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
E[3:0]		= Unchanged
U	•	= 0
AE	•	= 1

Description

The UNIMP_S instruction always raises an [Illegal Instruction](#) exception. The debugging tools can use this instruction to fill unused memory regions with an instruction that always remains unimplemented, regardless of future additions to the instruction set.

The program counter is not advanced before the exception is raised by this instruction. Therefore, on entry to the exception handler, the ERET register contains the address of the UNIMP_S instruction.

The status flags are not updated by this instruction.

Pseudo Code

```
InstError = 1;    /* UNIMP_S */
```

Assembly Code Example

```
UNIMP_S           ; Unimplemented Instruction
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

UNIMP_S	01111	001	111	00000

VADD2

Function

Dual 32-bit addition. For the DSP version, see [VADD2](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.w0 = B.w0 + C.w0;
    A.w1 = B.w1 + C.w1;}
```



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”](#).

Instruction Format

op A,B,C

Syntax Example

VADD2 A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing two 32-bit elements. These elements are normally register pairs, although the operand formats are not restricted to specifying registers. Each pair of elements from B and C is added to form two 32-bit sums. These are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if (cc)==true then                                     /* VADD2*/
    A.w0 = B.w0 + C.w0
    A.w1 = B.w1 + C.w1
```

Assembly Code Example

```
VADD2 r0,r2,r4    ; dual SIMD 32-bit addition of (r5,r4) and (r3,r2).
                  ;the result is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
VADD2	a,b,c	00101bbb001111000BBBCCCCCAAAAAA
VADD2	a,b,u6	00101bbb011111000BBBuuuuuuAAAAAA
VADD2	b,b,s12	00101bbb101111000BBBsssssssSSSSSS
VADD2<.cc>	b,b,c	00101bbb111111000BBBCCCCC0QQQQQ
VADD2<.cc>	b,b,u6	00101bbb111111000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VADD2H

Function

Dual 16-bit SIMD addition. For the DSP version, see [VADD2H](#)

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”](#).

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”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
VADD2H	a,b,c	00101bbb000101000BBBCCCCCAAAAAA
VADD2H	a,b,u6	00101bbb010101000BBBuuuuuuAAAAAA
VADD2H	b,b,s12	00101bbb100101000BBBssssssSSSSSS
VADD2H<.cc>	b,b,c	00101bbb110101000BBBCCCCC0QQQQQ
VADD2H<.cc>	b,b,u6	00101bbb110101000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VADD4H

Function

Quad 16-bit SIMD addition. For the DSP version, see [VADD4H](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.h0 = B.h0 + C.h0;
    A.h1 = B.h1 + C.h1;
    A.h2 = B.h2 + C.h2;
    A.h3 = B.h3 + C.h3;
}
```



Note

$h0, h1, h2, h3$ represent the 16-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"](#).

Instruction Format

op A,B,C

Syntax Example

VADD4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each pair of elements from B and C is added to form four 16-bit sums. These 16-bit sums are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if (cc) == true then                                     /* VADD4H*/
    A.h0 = B.h0 + C.h0
    A.h1 = B.h1 + C.h1
    A.h2 = B.h2 + C.h2
    A.h3 = B.h3 + C.h3
```

Assembly Code Example

```
VADD4H r0,r2,r4    ; quad SIMD 16-bit addition of (r3,r2) and (r5,r4). The
                   ;result is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
VADD4H	a,b,c	00101bbb001110000BBBCCCCCAAAAAA
VADD4H	a,b,u6	00101bbb011110000BBBuuuuuuAAAAAA
VADD4H	b,b,s12	00101bbb101110000BBBssssssSSSSSS
VADD4H<.cc>	b,b,c	00101bbb111110000BBBCCCCC0QQQQQ
VADD4H<.cc>	b,b,u6	00101bbb111110000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VADDSUB

Function

Dual 32-bit SIMD add and subtract. For the DSP version, see [VADDSUB](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.w0 = B.w0 + C.w0;
    A.w1 = B.w1 - C.w1;
}
```



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"](#).

Instruction Format

op A,B,C

Syntax Example

VADDSUB A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing two 32-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. The first pair of elements from B and C is added to form a 32-bit sum. The second pair of elements is subtracted to form a 32-bit difference. The sum and difference are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {
    A.w0 = B.w0 + C.w0;
    A.w1 = B.w1 - C.w1;} /* VADDSUB*/
```

Assembly Code Example

```
VADDSUB r0,r2,r4      ; dual SIMD 32-bit addition and subtraction of (r3,r2)
                      ;and (r5,r4). The result is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .
For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
VADDSUB	a,b,c	00101bbb001111100BBBCCCCCAAAAAA
VADDSUB	a,b,u6	00101bbb011111100BBBuuuuuuAAAAAA
VADDSUB	b,b,s12	00101bbb101111100BBBsssssssSSSSSS
VADDSUB<.cc>	b,b,c	00101bbb111111100BBBCCCCC0QQQQQ
VADDSUB<.cc>	b,b,u6	00101bbb111111100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VADDSUB2H

Function

Dual 16-bit SIMD add and subtract. For the DSP version, see [VADDSUB2H](#).

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit *b* and *c* operands specify vectors containing two 16-bit elements. The higher 16 bits from *b* and *c* is added to form a 16-bit sum. The lower 16 bits are subtracted to form a 16-bit difference. The sum and difference are assigned to the destination register, if defined.

This instruction does not modify any flags. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if(cc) {                                     /* VADDSUB2H*/
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```

Assembly Code Example

```
VADDSUB2H r1,r2,r3      ; dual SIMD 16-bit addition and subtraction
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
VADDSUB2H	a,b,c	00101bbb000101100BBBCCCCCAAAAAA
VADDSUB2H	a,b,u6	00101bbb010101100BBBuuuuuuAAAAAA
VADDSUB2H	b,b,s12	00101bbb100101100BBBsssssssSSSSSS
VADDSUB2H<.cc>	b,b,c	00101bbb110101100BBBCCCCC0QQQQQ
VADDSUB2H<.cc>	b,b,u6	00101bbb110101100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VADDSUB4H

Function

Quad 16-bit SIMD add and subtract. For the DSP version, see [VADDSUB4H](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.h0 = B.h0 + C.h0;
    A.h1 = B.h1 - C.h1;
    A.h2 = B.h2 + C.h2;
    A.h3 = B.h3 - C.h3;
}
```



Note

$h0$, $h1$, $h2$, and $h3$ represent the 16-bit elements of a 32-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"](#).

Instruction Format

op A,B,C

Syntax Example

VADDSUB4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each even-numbered pair of elements from B and C is added to form 16-bit sums. Each odd-numbered pair of elements from B and C is subtracted to form 16-bit differences. The four elemental results are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {                                     /* VADDSUB4H*/
    A.h0 = B.h0 + C.h0;
    A.h1 = B.h1 - C.h1;
    A.h2 = B.h2 + C.h2;
    A.h3 = B.h3 - C.h3;
}
```

Assembly Code Example

```
VADDSUB4H r0,r2,r4    ; quad SIMD 16-bit addition and subtraction of
                      ; (r3,r2) and (r5,r4). The result is stored in
                      ; (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
VADDSUB4H	a,b,c	00101bbb001110100BBBCCCCCAAAAAA
VADDSUB4H	a,b,u6	00101bbb011110100BBBuuuuuuAAAAAA
VADDSUB4H	b,b,s12	00101bbb101110100BBBssssssSSSSSS
VADDSUB4H<.cc>	b,b,c	00101bbb111110100BBBCCCCC0QQQQQ
VADDSUB4H<.cc>	b,b,u6	00101bbb111110100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VMAC2H

Signed multiplication and accumulation of two 16-bit vectors. For the DSP version, see [VMAC2H](#).

Function

Signed multiplication and accumulation of two 16-bit vectors.

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
A.w1 = acchi+(b.h1 * c.h1);
A.w0 = acclo+(b.h0 * c.h0);
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op A,b,c

Syntax Example

VMAC2H A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VMAC2H	a, b, c	00101 bbb 00 01111 00 BBB CCCCC AAAAAA
VMAC2H	a, b, u6	00101 bbb 01 01111 00 BBB uuuuuu AAAAAA
VMAC2H	b, b, s12	00101 bbb 10 01111 00 BBB ssssss SSSSSS
VMAC2H<.cc>	b, b, c	00101 bbb 11 01111 00 BBB CCCCC 0QQQQQ
VMAC2H<.cc>	b, b, u6	00101 bbb 11 01111 00 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VMAC2HU

This section describes the standard MPYD instruction. For the DSP version, see [VMAC2HU](#).

Function

Unsigned multiplication and accumulation of two 16-bit vectors.

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
A.w1 = acchi+(b.h1 * c.h1);
A.w0 = acclo+(b.h0 * c.h0);
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op A,b,c

Syntax Example

VMAC2HU A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

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) {
    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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VMAC2HU	a, b, c	00101bbb000111110BBBCCCCCAAAAAA
VMAC2HU	a, b, u6	00101bbb010111110BBBuuuuuuAAAAAA
VMAC2HU	b, b, s12	00101bbb100111110BBBssssssSSSSSS
VMAC2HU<.cc>	b, b, c	00101bbb110111110BBBCCCCC0QQQQQ
VMAC2HU<.cc>	b, b, u6	00101bbb110111110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VMPY2H

This section describes the standard VMPY2H instruction. For the DSP version, see [VMPY2H](#).

Function

Dual 16-bit SIMD multiplication

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
if (cc) {
    acc.w0 = A.w0 = b.h0 * c.h0;
    acc.w1 = A.w1 = b.h1 * c.h1;
}
```



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"](#).

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

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
VMPY2H	a,b,c	00101bbb000111000BBBCCCCCAAAAAA
VMPY2H	a,b,u6	00101bbb010111000BBBuuuuuuAAAAAA
VMPY2H	b,b,s12	00101bbb100111000BBBssssssSSSSSS
VMPY2H<.cc>	b,b,c	00101bbb110111000BBBCCCCC0QQQQQ
VMPY2H<.cc>	b,b,u6	00101bbb110111000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VMPY2HU

This section describes the standard MPYD instruction. For the DSP version, see [VMPY2HU](#).

Function

Dual unsigned 16-bit SIMD multiplication

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
if (cc) {
    acc.w0 = A.w0 = b.h0 * c.h0;
    acc.w1 = A.w1 = b.h1 * c.h1;
}
```



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”](#).

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

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .
For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
VMPY2HU	a,b,c	00101bbb0001111010BBBCCCCCAAAAAA
VMPY2HU	a,b,u6	00101bbb0101111010BBBuuuuuuAAAAAA
VMPY2HU	b,b,s12	00101bbb1001111010BBBsssssssSSSSSS
VMPY2HU<.cc>	b,b,c	00101bbb1101111010BBBCCCCC0QQQQQ
VMPY2HU<.cc>	b,b,u6	00101bbb1101111010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VSUB2

Function

Dual 32-bit SIMD subtract. For the DSP version, see [VSUB2](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.w0 = B.w0 - C.w0;
    A.w1 = B.w1 - C.w1;
}
```



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”](#).

Instruction Format

op A,B,C

Syntax Example

VSUB2 A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing two 32-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each pair of elements from B and C is subtracted to form two 32-bit differences. These differences are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) { /* VSUB2*/
    A.w0 = B.w0 - C.w0;
    A.w1 = B.w1 - C.w1;}
```

Assembly Code Example

```
VSUB2 r0,r2,r4 ; dual SIMD 32-bit subtraction of
                ;(r3,r2) and (r5,r4). The result
                ;is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)” . For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)” .

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)” .

Instruction Code		
VSUB2	a,b,c	00101bbb001111010BBBCCCCCAAAAAA
VSUB2	a,b,u6	00101bbb011111010BBBuuuuuuAAAAAA
VSUB2	b,b,s12	00101bbb101111010BBBssssssSSSSSS
VSUB2<.cc>	b,b,c	00101bbb111111010BBBCCCCC0QQQQQ
VSUB2<.cc>	b,b,u6	00101bbb111111010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VSUB2H

Function

Dual 16-bit vector subtraction. For the DSP version, see [VSUB2H](#).

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Each pair of elements from b and c is subtracted to form two 16-bit differences. These differences are assigned to the destination register, if defined.

This instruction does not modify any flags. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if(cc) {                                     /* VSUB2H*/
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 - c.h1;
}
```

Assembly Code Example

```
VSUB2H r1,r2,r3                          ; dual SIMD 16-bit subtraction
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUB2H	a, b, c	00101 bbb 00 0101010 BBBCCCCC AAAAAA
VSUB2H	a, b, u6	00101 bbb 01 0101010 BBBuuuuuu AAAAAA
VSUB2H	b, b, s12	00101 bbb 10 0101010 BBBssssss SSSSSS
VSUB2H<.cc>	b, b, c	00101 bbb 11 0101010 BBBCCCCC 0QQQQQ
VSUB2H<.cc>	b, b, u6	00101 bbb 11 0101010 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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VSUB4H

Function

Quad 16-bit SIMD subtraction. For the DSP version, see [VSUB4H](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.h0 = B.h0 - C.h0;
    A.h1 = B.h1 - C.h1;
    A.h2 = B.h2 - C.h2;
    A.h3 = B.h3 - C.h3;
}
```



Note

h0, h1, h2, and h3 represent the 16-bit elements of a 32-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"](#).

Instruction Format

op A,B,C

Syntax Example

VSUB4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each pair of elements from B and C is subtracted to form four 16-bit differences. These differences are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {      A.h0 = B.h0 - C.h0;                /* VSUB4H*/
  A.h1 = B.h1 - C.h1;
  A.h2 = B.h2 - C.h2;
  A.h3 = B.h3 - C.h3;
}
```

Assembly Code Example

```
VSUB4H r0,r2,r4      ; quad SIMD 16-bit subtraction
                     ; (r3,r2) and (r5,r4). The result
                     ; is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
VSUB4H	a,b,c	00101bbb001110010BBBCCCCCAAAAAA
VSUB4H	a,b,u6	00101bbb011110010BBBuuuuuuAAAAAA
VSUB4H	b,b,s12	00101bbb101110010BBBssssssSSSSSS
VSUB4H<.cc>	b,b,c	00101bbb111110010BBBCCCCC0QQQQQ
VSUB4H<.cc>	b,b,u6	00101bbb111110010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VSUBADD

Function

Dual 32-bit SIMD subtract and add. For the DSP version, see [VSUBADD](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.w0 = B.w0 - C.w0;
    A.w1 = B.w1 + C.w1;
}
```



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"](#).

Instruction Format

op A,B,C

Syntax Example

VSUBADD A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing two 32-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. The first pair of elements from B and C is subtracted to form a 32-bit difference. The second pair of elements is added to form a 32-bit sum. The difference and sum are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {
    A.w0 = B.w0 - C.w0;
    A.w1 = B.w1 + C.w1;
}
```

/* VSUBADD*/

Assembly Code Example

```
VSUBADD r0,r2,r4    ; dual SIMD 32-bit subtraction ;and addition of (r3,r2)
                   ;and (r5,r4). The result is stored in (r1,r0)
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUBADD	a, b, c	00101bbb001111110BBBCCCCCAAAAAA
VSUBADD	a, b, u6	00101bbb011111110BBBuuuuuuAAAAAA
VSUBADD	b, b, s12	00101bbb101111110BBBssssssSSSSSS
VSUBADD<.cc>	b, b, c	00101bbb111111110BBBCCCCC0QQQQQ
VSUBADD<.cc>	b, b, u6	00101bbb111111110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VSUBADD2H

Function

Dual 16-bit vector subtraction and addition. For the DSP version, see [VSUBADD2H](#).

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 + c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUBADD2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit `b` and `c` operands specify vectors containing two 16-bit elements. The lower 16-bit elements from `b` and `c` are subtracted to form a 16-bit difference. The higher 16-bit elements are added to form a 16-bit sum. The difference and sum are assigned to the destination register.

This instruction does not modify any flags. An illegal instruction exception is raised if `LP_COUNT` (r60) is given as the destination register.

Pseudo Code

```
if(cc) {                                                    /* 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

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code		
VSUBADD2H	a,b,c	00101bbb000101110BBBCCCCCAAAAAA
VSUBADD2H	a,b,u6	00101bbb010101110BBBuuuuuuAAAAAA
VSUBADD2H	b,b,s12	00101bbb100101110BBBssssssSSSSSS
VSUBADD2H<.cc>	b,b,c	00101bbb110101110BBBCCCCC0QQQQQ
VSUBADD2H<.cc>	b,b,u6	00101bbb110101110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are

unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section [“Dual-operand Instructions, F32_GEN4”](#).

VSUBADD4H

Function

Quad16-bit SIMD subtract and add. For the DSP version, see [VSUBADD4H](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.h0 = B.h0 - C.h0;
    A.h1 = B.h1 + C.h1;
    A.h2 = B.h2 - C.h2;
    A.h3 = B.h3 + C.h3;
}
```



Note

h0, h1, h2, and h3 represent the 16-bit elements of a 32-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"](#).

Instruction Format

op a, b, c

Syntax Example

VSUBADD4H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each odd-numbered pair of elements from B and C is added to form 16-bit sums. Each even-numbered pair of elements from B and C is subtracted to form 16-bit differences. The four elemental results are assigned to the destination register, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if (cc) ==true then                                /* VSUBADD4H*/
    A.h0 = B.h0 - C.h0
    A.h1 = B.h1 + C.h1
    A.h2 = B.h2 - C.h2
    A.h3 = B.h3 + C.h3
```

Assembly Code Example

```
VSUBADD4H r0,r2,r4    ; quad SIMD 16-bit subtraction and addition of r2
                      ;and r4
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#) . For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#) .

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#) .

Instruction Code

VSUBADD4H	a,b,c	00101bbb001110110BBBCCCCCAAAAAA
VSUBADD4H	a,b,u6	00101bbb011110110BBBuuuuuuAAAAAA
VSUBADD4H	b,b,s12	00101bbb101110110BBBssssssSSSSSS
VSUBADD4H<.cc>	b,b,c	00101bbb111110110BBBCCCCC0QQQQQ
VSUBADD4H<.cc>	b,b,u6	00101bbb111110110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

WEVT

Function

Enter the sleep state and wait for an event

Extension Group

BASELINE

Operation

DEBUG[ZZ] = 1;

Instruction Format

op c

Syntax Example

WEVT

STATUS32 Flags Affected

IE	• Updated depending on the SLEEP operand as explained in Table 9-6
E[3:0]	• Updated depending on the SLEEP operand as explained in Table 9-6
Z	= Unchanged
N	= Unchanged
C	= Unchanged
V	= Unchanged
ZZ	• = 1
SM	• Updated depending on the SLEEP operand as explained in Table 9-6

Description

The ARCV2-based processor enters the sleep state when the processor encounters the WEVT instruction. The processor stays in the sleep state until one of the following events occurs:

- ❑ An external event input is asserted; the external input transitions from 0 to 1.
- ❑ An interrupt is taken.
- ❑ An external halt request is acknowledged by the processor.
- ❑ A debug transaction places the processor in a halt state.

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”](#).

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 9-7 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

WEVT	c	0010000001011110001CCCCC111111
WEVT	<u6>	0010000011011110001uuuuuu111111

WLFC

Function

Enter the sleep state to reduce dynamic power during busy-waiting loops

Extension Group

ATOMIC_OPTION == 1.

Operation

DEBUG[ZZ] = 1;

Instruction Format

op c

Syntax Example

WLFC

STATUS32 Flags Affected

IE	• Updated depending on the SLEEP operand as explained in Table 9-6
E[3:0]	• Updated depending on the SLEEP operand as explained in Table 9-6
Z	= Unchanged
N	= Unchanged
C	= Unchanged
V	= Unchanged
ZZ	• = 1
SM	• Updated depending on the SLEEP operand as explained in Table 9-6

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”](#).

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 9-8 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”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
WLFC	c	00100001001011110001CCCCC111111
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, L IMM

op a, L IMM, b

Syntax Example

XBFU<.f> b, b, u5, u5

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Source operand 1 specifies a 32-bit value, from which a bit-field is extracted and assigned to the destination operand. The extracted bit-field starts at bit *N* and includes *M* bits of operand 1.

The value of *N* is specified by the unsigned binary value obtained from the least-significant five bits of operand 2. *M* is obtained by adding 1 to the unsigned five-bit binary value obtained from bits [9:5] of operand 2.

The first source operand is shifted right, without sign-extension, by *N*, and then logically ANDed with a mask of size *M*+1 bits. The result is written into the destination register *a*. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

N = src2[4:0]
M = src2[9:5] + 1
if cc==true then
    dest = (src1 >> N) && ((1 << M)-1)
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
/* XBFU
*first bit to extract
*number of bits to
*extract
*/

```

Assembly Code Example

```

XBFU r1,r1,4,6      ;extract bits 4 through 9 from r1 into r1

XBFU r2,r2,0,5      ; extract bits 0 through 5 from r2 into r2
XBFU r1,r2,r3        ;extract bits from r2 depending on r3
XBFU.F 0,r2,3,5      ;set Z flag if bits 3 to 7 in r2 are zero

```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
XBFU<.f>	a,b,c	00100bbb00101101FBBBCCCCCAAAAAA
XBFU<.f>	a,b,u6	00100bbb01101101FBBBuuuuuuAAAAAA
XBFU<.f>	b,b,s12	00100bbb10101101FBBBssssssSSSSSS
XBFU<.cc><.f>	b,b,c	00100bbb11101101FBBBCCCCC0QQQQQ
XBFU<.cc><.f>	b,b,u6	00100bbb11101101FBBBuuuuuu1QQQQQ

If the first source and destination operand are the same register, an XBFU instruction can be encoded using the b, b, s12 operand format, allowing two five-bit constant values to be encoded within the s12 second operand.

If the destination register must be distinct from the first source operand, the two five-bit offsets of the XBFU instruction can be encoded using a LMM value as the second source operand, that is, using format a, b, c where c is a LMM.

An XBFU instruction that extracts a field that is less than or equal to two bits can be encoded using a u6 operand. In this case, only bit 0 of the M operand can be specified, and bits 1 through 4 are assumed to be zero. The N operand can be fully-specified as bits 0 through 4 of the u6 operand. This format allows the source and destination operand registers to be distinct.

op b, b, u6, u6

XOR

Function

Bitwise Exclusive OR

Extension Group

BASELINE

Operation

if (cc) $a = b \wedge c$;

Instruction Format

op a, b, c

Syntax Example

XOR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise exclusive OR of source operand 1 (b) with source operand 2 (c). The result is written into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```

if cc==true then                                /* XOR */
    dest = src1 XOR src2
    if F==1 then
        Z_flag = if dest==0 then 1 else 0
        N_flag = dest[31]

```

Assembly Code Example

```
XOR r1,r2,r3 ; Logical XOR contents of r2 with r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
XOR<.f>	a, b, c	00100bbb00000111FBBBCCCCC(AAAAAA)
XOR<.f>	a, b, u6	00100bbb01000111FBBBuuuuuu(AAAAAA)
XOR<.f>	b, b, s12	00100bbb10000111FBBBssssss(SSSSSS)
XOR<.cc><.f>	b, b, c	00100bbb11000111FBBBCCCCC(0QQQQQ)
XOR<.cc><.f>	b, b, u6	00100bbb11000111FBBBuuuuuu(1QQQQQ)
XOR_S	b, b, c	01111bbbccc00111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section [“Dual-operand Instructions, F32_GEN4”](#).

10

The Host

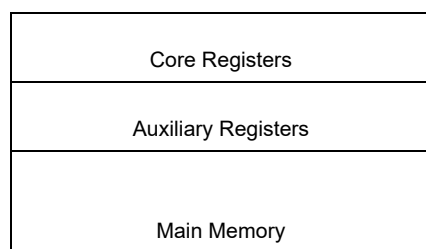
10.1 The Host Interface

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.

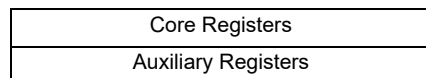
Registers and the program memory of ARCV2-based processor appear as a memory mapped section to the host. [Figure 10-1](#) and [Figure 10-2](#) show an example host system using contiguous part of host memory and an example host system using a section of memory and a section of I/O space, respectively.

Figure 10-1 Example Host Memory Maps, Contiguous Host Memory

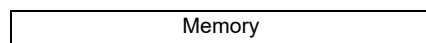


Host I/O Map

Figure 10-2 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 10-1](#) on page 850.

Table 10-1 Host Accesses to the ARCV2-based processor

	Running	Halted
Memory	Read/Write	Read/Write
Auxiliary Registers	Read/Write	Read/Write
Core Registers	Read/Write	Read/Write

10.2 Core and Auxiliary Registers

Any debug operation can be carried out while the processor is running. The debug port inserts a pseudo-instruction into the processor pipeline to carry out the requested operation. It is therefore intrusive to normal program execution if carried out while the processor is running (that is, the debug operation steals instruction cycles from the application that is running).

10.3 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.

10.4 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.

10.5 Starting

The host starts the ARCV2-based processor by clearing the H bit in the [Status Register, STATUS32](#) register.

If the ARCV2-based processor is running code, and needs to be restarted at a different location, you must put the processor into a state similar to the post-[Reset](#) condition to ensure correct operation. The procedure is as follows:

1. Reset the three hardware loop registers to their default values.
2. Disable interrupts, using the status register.
3. Any extension logic must be reset to its default state.
4. Host sets PC to the desired restart location.

If the ARCV2-based processor is running and needs to be restarted to *continue* where the processor left off, the procedure is as follows:

1. The host reads the status from the STATUS Register (see [Status Register, STATUS32](#)).
2. The host writes back to the STATUS32 register (see [Status Register, STATUS32](#)) with *the same* value as was just read, but clearing the H bit.
3. The ARCV2-based processor continues from where it left off when it was stopped.

10.6 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 10-2](#) on page 851).

Table 10-2 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.

10.6.1 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 760 and [“Null Instruction Format”](#) on page 122 for further details.

10.6.2 BRK Instruction in Single Instruction Step Mode

The [BRK](#) instruction behaves exactly as when the processor is not in the Single Step Mode.

10.7 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

Memory and System Components

In this part:

- “Protection Schemes”
- “Memory Management Unit”
- “ICCM”
- “Instruction Cache”
- “DCCM”
- “Data Cache”
- “Cluster Shared Memory”
- “External Host Debugging”
- “Debug Features”
- “Performance Counters”
- “Processor Timers”
- “Error Protection”
- “Multi-core Cluster”
- “L2 Cache”
- “Shared Memory Configuration”
- “Loop Buffer”

- “Shared Coherency Unit”
- “Peripheral Bus”
- “Branch Prediction Unit”
- “Power Management Features”
- “Floating-Point Unit”
- “User Auxiliary Register Interface”
- “Debug Features”
- “ARConnect”
- “SmaRT”
- “Write Buffer”
- “ROM Patching Unit”

11

Memory Protection Unit

11.1 Memory Protection Unit

The ARCV2-based Memory Protection Unit (MPU) provides protection by dividing the address space into regions associated with specific attributes such as Read, Write, and Execute. If an attempt is made to access a region for which an associated attribute is not permitted, the ARCV2-based processor raises a Protection Violation exception, and this exception prevents the faulting instruction from completing.

The ARCV2-based processor provides kernel-mode capabilities such as separate Read, Write, and Execute permissions for the user and kernel mode processes on each memory region.

The provision of distinct kernel and user permissions allows an operating system to protect its code and data from illegal or unexpected accesses by user-mode processes. A kernel-mode process is always permitted to access any region for which the required permission is provided to user-mode processes.

You can define default kernel and user permissions for accesses that fall outside all memory protection regions.

Multiple MPU regions are allowed to overlap; the attributes that apply to a particular address in the case of overlapping regions are determined using a fixed priority scheme.

11.1.1 Key Features

The following are the main features of the ARCV2-based processor MPU:

- Programmable execute permission bits to enable or disable execution of code from specific regions of memory
- Programmable data read and write permission bits to enable or disable data access to specific regions of memory
- Separate kernel and user mode read, write, and execute permissions
- 8, 16, 18, 20, 22, 24, 26, 28, 30, or 32 configurable memory regions
- Regions that can be programmed individually and independently
- Overlapping regions supported by a priority scheme
- Programmable default permissions that apply to accesses outside all programmed protection regions
- EV_ProtV (see [Exception Vectors and the Exception Cause Register](#) on page 228) exception to indicate access violations
- Protection exceptions that are precise and can be restarted
- Designed to be used in conjunction with the Stack Checking and/or Code Protection mechanisms in the ARCV2-based processors
- Configurable cacheable or uncacheable regions

11.1.2 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.

11.1.3 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](#).

11.1.4 Data Organization and Addressing

The ARCV2 address space is divided into 16 equal-sized regions that are used to define the locations of memories (ICCM and DCCM), as well as the location of peripherals and uncacheable memory.

The ARCV2 MPU allows software to further define a number of variable-sized protection regions with programmable attributes that control what types of reference are permitted in each protection region. The maximum number of protection regions is fixed at configuration time and is specified in the MPU_BUILD register ([Memory Protection Build Configuration Register, MPU_BUILD](#)).

Protection regions are allowed to overlap each other: the attributes that apply to a particular address in the case of overlapping regions are determined using a fixed priority scheme. This mechanism can be used to implement a flexible organization based on need, for example, a large read-only region with a smaller embedded writable region.

Each region has a base address and size, specified in the [MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB31](#) and [MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP31](#). 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:

11.1.4.1 Example 1 - Contiguous Regions

In this example, the memory has the following features:

- A user-executable region in the first half of memory
- A user-readable and user-writable region in the next quarter of memory
- A user-readable and user-writable region for 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 11-1 Example 1 -- Contiguous Regions Memory Map

Region 0	0x00000000	Base address: 0x00000000
	0x7FFFFFFF	Region size: 2G
Region 1	0x80000000	Regions permissions: user-execute
	0xBFFFFFFF	Base address: 0x80000000
Region 2	0xC0000000	Region size: 1G
	0xFFFFFFF	Regions permissions: user-read, user-write
	0xC0000000	Base address: 0xC0000000
	0xFFFFFFF	Region size: 1G
	0xFFFFFFF	Regions permissions: user-read, user-write

The base address and size, as specified in the MPU Region Descriptor Base Registers and MPU Region Descriptor Permissions Registers, is set up as shown in the following figures:

Figure 11-1 Example 1: MPU_RDB0 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
BASE_ADDRESS																												RESERVED				V
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Figure 11-2 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																		DC	IC	SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	1	0

Figure 11-3 Example 1: MPU_RDB1 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
BASE_ADDRESS																												RESERVED				V
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Figure 11-4 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																		DC	IC	SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1

Figure 11-5 Example 1: MPU_RDB2 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
BASE_ADDRESS																												RESERVED				V
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Figure 11-6 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																		DC	IC	SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	1

11.1.4.2 Example 2 - Overlapping Regions

In this example the memory has the following features:

- The entire memory space is defined as user-readable and user-writable using the lowest priority region (for an 8-region setup this region is region 7).
- There is then an overlapping user-accessible, execute-only region in the lower 1G of memory to contain the vector table and kernel operating system code.

The memory map for this setup is shown in the following figure:

Figure 11-7 Overlapping Regions Memory Map

Region 0 overlaps with Region 7	0x00000000	Base address: 0x00000000 Region size: 1G Region permissions: user- execute	
	0x3FFFFFFF		
Region 7	0x40000000		Base address: 0x00000000 Region size: 4G Region permissions: user-read, user-write
	0xFFFFFFFF		

The base address and size, as specified in the MPU Region Descriptor Base Registers and MPU Region Descriptor Permissions Registers, are set up as shown in the following figures:

Figure 11-8 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 11-9 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																		DC	IC	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 11-10 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 11-11 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																		DC	IC	SIZE[4:2]			KR	KW	KE	UR	UW	UE	R	SIZE	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	1	1

11.1.5 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](#)

11.1.5.1 Region Accesses

All accesses are checked against the region descriptors, according to a fixed priority, to determine what attributes to apply.

Table 11-2 Priority Configurations

Region	Priority (8-region configuration)	Priority (16-region configuration)	Priority (32-region configuration)
0	Highest	Highest	Highest
1	2 nd Highest	2 nd Highest	2 nd Highest
2	3 rd Highest	3 rd Highest	3 rd Highest
3	4 th Highest	4 th Highest	4 th Highest
4	5 th Highest	5 th Highest	5 th Highest
5	6 th Highest	6 th Highest	6 th Highest
6	7 th Highest	7 th Highest	7 th Highest
7	Lowest	8 th Highest	8 th Highest
8	Not applicable	9 th Highest	9 th Highest
9	Not applicable	10 th Highest	10 th Highest
10	Not applicable	11 th Highest	11 th Highest
11	Not applicable	12 th Highest	12 th Highest
12	Not applicable	13 th Highest	13 th Highest
13	Not applicable	14 th Highest	14 th Highest
14	Not applicable	15 th Highest	15 th Highest
15	Not applicable	Lowest	16 th Highest

Region	Priority (8-region configuration)	Priority (16-region configuration)	Priority (32-region configuration)
16	Not applicable	Not applicable	17 th Highest
17	Not applicable	Not applicable	18 th Highest
18	Not applicable	Not applicable	19 th Highest
19	Not applicable	Not applicable	20 th Highest
20	Not applicable	Not applicable	21 th Highest
21	Not applicable	Not applicable	22 th Highest
22	Not applicable	Not applicable	23 th Highest
23	Not applicable	Not applicable	24 th Highest
24	Not applicable	Not applicable	25 th Highest
25	Not applicable	Not applicable	26 th Highest
26	Not applicable	Not applicable	27 th Highest
27	Not applicable	Not applicable	28 th Highest
28	Not applicable	Not applicable	29 th Highest
29	Not applicable	Not applicable	30 th Highest
30	Not applicable	Not applicable	31 th Highest
31	Not applicable	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.

11.1.5.2 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 11-3](#) lists the settings of the ECR register based on the entry to the EV_ProtV exception.

Table 11-3 Setting ECR Register based on EV_ProtV Exception

Failed Reference type	Vector Name	Vector Offset	Vector Number	Cause Code	Parameter	ECR value
Memory Read (such as LD, POP, LEAVE, interrupt exit, LLOCK)	EV_ProtV	0x18	0x06	0x01	0x04	0x060104
Memory Write (such as ST, PUSH, ENTER, interrupt entry, SCOND)	EV_ProtV	0x18	0x06	0x02	0x04	0x060204
Memory Read-Modify-Write (such as EX)	EV_ProtV	0x18	0x06	0x03	0x04	0x060304

Bit 2 of the 8-bit parameter field is set to 1 to indicate that a protection violation was reported by the MPU.

The exception vector table and the EV_ProtV exception handler must always be in a valid region with at least kernel execute permission. If they are not, the processor raises a double exception on accessing the exception vector table or the handler and triggers a non-restartable Machine Check exception.

When execution enters the exception handler, the address that caused the protection violation is available in the [Exception Fault Address, EFA](#) (0x404).

In the case of instruction-fetch violations, the EFA register is set to the address of the instruction fetch that caused the violation. Care must be taken to ensure that the instruction or data elements do not straddle MPU region boundaries as access protection checks are performed only using the starting address (first byte at the lowest address) of the element. In the case of data access violations, EFA is set to the address of the data access that caused the violation.

**Note**

When arranging MPU regions and user code, care must be taken to ensure that when a branch has a delay slot, both the branch and its delay slot always reside within a single MPU region or in regions that have the same set of access permissions.

11.1.5.3 Multiple Simultaneous Protection Violations

In a processor with a precise exception model, the first violation in program order is the one that must be taken. This model requires that all outstanding instructions that have already been committed must complete and all uncommitted instructions must be discarded. Faulting instructions may then be retried after the violation had been dealt with.

In ARCV2-based processors, all protection violation exceptions are precise. Therefore, all non-violating instructions that are issued before a protection violation are detected and allowed to complete, and the instruction that raises the protection violation is dismissed. Any instructions that occur after the faulting instruction, and which themselves may cause protection violations, are also dismissed when the faulting instruction is dismissed.

In the case of simultaneous instruction and data violations generated by a single instruction, the instruction fetch violation is reported and the data violation is ignored as ARCV2 architecture gives higher priority to instruction fetch violations.

Due to the pipelined construction of the ARCV2-based processor, the instruction fetch and data access violations are detected by hardware at the same time, even though the data violation arises from an earlier instruction than the instruction fetch violation. In this case, the data violation takes precedence, as the violation occurs earlier in program order.

Following are the potential sources of a protection violation in the ARCV2 architecture: Code Protection, Stack Checking, MPU, and MMU. The MPU and MMU options are mutually exclusive, and therefore exceptions from these two sources cannot occur simultaneously. However, you can configure an ARCV2-based processor with MPU, Stack Checking, and Code Protection, and so one Load or Store instruction to violate the protection policies of all three schemes simultaneously. As Stack Checking and Code Protection policies apply only to Load or Store types of references, only the MPU (or MMU) can raise protection violations in relation to instruction fetches.

Each source of protection violation independently sets a different bit in the parameter field of the Exception Cause Register (ECR), allowing multiple simultaneous violations to be reported by the same exception. For reference they are summarized in the following table:

Table 11-4 ECR parameter values for EV_ProtV sources

Source of EV_ProtV	ECR Parameter Value
Code protection	0x01
Stack Checking	0x02
MPU	0x04
MMU	0x08

Therefore, if the address of a POP instruction is detected by the MPU as a protection violation, as well as having an address that is outside the defined stack region, and also reads from a code area that is protected by the Code Protection mechanism, the ECR parameter value is 0x7 (MPU, Stack Checking, and Code Protection bits all set).

However, if the same instruction also raises a protection violation due to its PC value being within a region that does not have execute permission, the instruction violation takes precedence over all data violations from whatever source, thus the ECR parameter value is 0x4 (MPU only).

11.1.5.4 Caution When Updating Region Settings

As the MPU protects all the memory space on an ARCV2-based processor, care must be taken when updating its region descriptors to ensure that both the exception vector space and the code modifying the region descriptors remain executable, even during the update. Therefore, you must disable the MPU before any region descriptors are updated by setting the MPU_EN register to 0x00000000. After all the region descriptors have been set up appropriately, the MPU can be re-enabled by setting MPU_EN to 0x40000000. The instruction immediately following an AUX write enabling the MPU should not reside in a no-execute MPU region.

Similarly, when you modify the region descriptors through the debugger via host access when the CPU is running, you must disable the MPU before any region descriptors are updated by setting the MPU_EN register to 0x00000000.

The debugger is never prevented by the MPU from accessing memory at any position in the memory map.

If a region descriptor is programmed with reserved values in the size field, the behavior of that region is undefined if the region and the MPU are both enabled. In practice, this kind of programming may lead to unexpected EV_ProtV violations being reported, or to the absence of such reports from any such incorrectly programmed MPU region. Therefore, use only legal values in the region descriptors when they are enabled. A disabled region descriptor never triggers a protection violation and can be programmed with illegal values without causing problems.

11.1.6 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 11-5](#) lists the MPU auxiliary registers.

Table 11-5 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 (Bank 0)	MPU region descriptor base 0
0x422	MPU_RDB16 (Bank 1)	MPU region descriptor base 16
0x423	MPU_RDP0 (Bank 0)	MPU region descriptor permissions 0
0x423	MPU_RDP16 (Bank 1)	MPU region descriptor permissions 16
0x424	MPU_RDB1 (Bank 0)	MPU region descriptor base 1
0x424	MPU_RDB17 (Bank 1)	MPU region descriptor base 17
0x425	MPU_RDP1 (Bank 0)	MPU region descriptor permissions 1
0x425	MPU_RDP17 (Bank 1)	MPU region descriptor permissions 17

Table 11-5 MPU Register Set

Address	Auxiliary Register Name	Description
0x426	MPU_RDB2 (Bank 0)	MPU region descriptor base 2
0x426	MPU_RDB18 (Bank 1)	MPU region descriptor base 18
0x427	MPU_RDP2 (Bank 0)	MPU region descriptor permissions 2
0x427	MPU_RDP18 (Bank 1)	MPU region descriptor permissions 18
0x428	MPU_RDB3 (Bank 0)	MPU region descriptor base 3
0x428	MPU_RDB19 (Bank 1)	MPU region descriptor base 19
0x429	MPU_RDP3 (Bank 0)	MPU region descriptor permissions 3
0x429	MPU_RDP19 (Bank 1)	MPU region descriptor permissions 19
0x42a	MPU_RDB4 (Bank 0)	MPU region descriptor base 4
0x42a	MPU_RDB20 (Bank 1)	MPU region descriptor base 20
0x42b	MPU_RDP4 (Bank 0)	MPU region descriptor permissions 4
0x42b	MPU_RDP20 (Bank 1)	MPU region descriptor permissions 20
0x42c	MPU_RDB5 (Bank 0)	MPU region descriptor base 5
0x42c	MPU_RDB21 (Bank 1)	MPU region descriptor base 21
0x42d	MPU_RDP5 (Bank 0)	MPU region descriptor permissions 5
0x42d	MPU_RDP21 (Bank 1)	MPU region descriptor permissions 21
0x42e	MPU_RDB6 (Bank 0)	MPU region descriptor base 6
0x42e	MPU_RDB22 (Bank 1)	MPU region descriptor base 22
0x42f	MPU_RDP6 (Bank 0)	MPU region descriptor permissions 6
0x42f	MPU_RDP22 (Bank 1)	MPU region descriptor permissions 22
0x430	MPU_RDB7 (Bank 0)	MPU region descriptor base 7
0x430	MPU_RDB23 (Bank 1)	MPU region descriptor base 23
0x431	MPU_RDP7 (Bank 0)	MPU region descriptor permissions 7
0x431	MPU_RDP23 (Bank 1)	MPU region descriptor permissions 23
0x432	MPU_RDB8 (Bank 0)	MPU region descriptor base 8
0x432	MPU_RDB24 (Bank 1)	MPU region descriptor base 24
0x433	MPU_RDP8 (Bank 0)	MPU region descriptor permissions 8
0x433	MPU_RDP24 (Bank 1)	MPU region descriptor permissions 24

Table 11-5 MPU Register Set

Address	Auxiliary Register Name	Description
0x434	MPU_RDB9 (Bank 0)	MPU region descriptor base 9
0x434	MPU_RDB25 (Bank 1)	MPU region descriptor base 25
0x435	MPU_RDP9 (Bank 0)	MPU region descriptor permissions 9
0x435	MPU_RDP25 (Bank 1)	MPU region descriptor permissions 25
0x436	MPU_RDB10 (Bank 0)	MPU region descriptor base 10
0x436	MPU_RDB26 (Bank 1)	MPU region descriptor base 26
0x437	MPU_RDP10 (Bank 0)	MPU region descriptor permissions 10
0x437	MPU_RDP26 (Bank 1)	MPU region descriptor permissions 26
0x438	MPU_RDB11 (Bank 0)	MPU region descriptor base 11
0x438	MPU_RDB27 (Bank 1)	MPU region descriptor base 27
0x439	MPU_RDP11 (Bank 0)	MPU region descriptor permissions 11
0x439	MPU_RDP27 (Bank 1)	MPU region descriptor permissions 27
0x43a	MPU_RDB12 (Bank 0)	MPU region descriptor base 12
0x43a	MPU_RDB28 (Bank 1)	MPU region descriptor base 28
0x43b	MPU_RDP12 (Bank 0)	MPU region descriptor permissions 12
0x43b	MPU_RDP28 (Bank 1)	MPU region descriptor permissions 28
0x43c	MPU_RDB13 (Bank 0)	MPU region descriptor base 13
0x43c	MPU_RDB29 (Bank 1)	MPU region descriptor base 29
0x43d	MPU_RDP13 (Bank 0)	MPU region descriptor permissions 13
0x43d	MPU_RDP29 (Bank 1)	MPU region descriptor permissions 29
0x43e	MPU_RDB14 (Bank 0)	MPU region descriptor base 14
0x43e	MPU_RDB30 (Bank 1)	MPU region descriptor base 30
0x43f	MPU_RDP14 (Bank 0)	MPU region descriptor permissions 14
0x43f	MPU_RDP30 (Bank 1)	MPU region descriptor permissions 30
0x440	MPU_RDB15 (Bank 0)	MPU region descriptor base 15
0x440	MPU_RDB31 (Bank 1)	MPU region descriptor base 31
0x441	MPU_RDP15 (Bank 0)	MPU region descriptor permissions 15
0x441	MPU_RDP31 (Bank 1)	MPU region descriptor permissions 31

11.1.7 MPU Enable Register, MPU_EN

Address: 0x409

Access: RW

Reset: 0x0000_0000

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.

This register contains 0x00000000 on reset.

Figure 11-12 MPU_EN Register

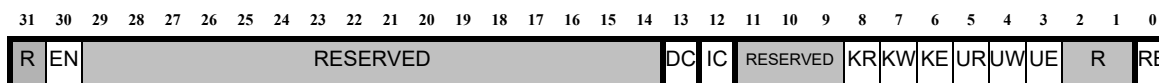


Table 11-6 MPU Enable Register

Field	Bit	Description
RB	[0]	Region bank 0x0: Bank for regions 0 to 15 0x1: Bank for regions 16 to 31 Notes: This bit is programmable only when the MPU version $\geq$ 0x06. For other versions of MPU, this bit is read as zero and ignored on writes. This bit can be set to 1 even if you have configured 16 or less than 16 regions. However, when you program the base and descriptor registers of the unavailable regions, an instruction error exception is raised.
UE	[3]	General Execute Permission 0: Instruction fetch is not permitted in the default protection region in user mode. 1: Instruction fetch is permitted in the default protection region in user or kernel mode (irrespective of the KE bit value).

Table 11-6 MPU Enable Register

Field	Bit	Description
UW	[4]	General Write Permission 0: Default protection region is write-protected in user mode. 1: Default protection region is writable when in user or kernel mode (irrespective of the KW bit value).
UR	[5]	General Read Permission 0: Default protection region is read-protected in user mode. 1: Default protection region is readable in user or kernel mode (irrespective of the KR bit value). Note that the UR field controls only read access; instruction-fetch access is controlled by the UE field.
KE	[6]	Kernel-only Execute Permission 0: Instruction fetch is not permitted in default protection region. 1: Instruction fetch is permitted to the default protection region when in kernel mode only.
KW	[7]	Kernel-only Write Permission 0: Default protection region is write-protected. 1: Default protection region is writable when in kernel mode only.
KR	[8]	Kernel-only Read Permission 0: Default protection region is read-protected. 1: Default protection region is readable when in kernel mode only. Note that the KR field controls only read access; instruction-fetch access is controlled by the KE field.

Table 11-6 MPU Enable Register

Field	Bit	Description
IC	[12]	Global code cacheability that applies to accesses outside all protected regions. 0x0: Code is cacheable in all levels of the cache hierarchy. 0x1: Code is not cacheable in any level of the cache hierarchy (L1, L2, and so on). Notes: - This bit is valid only when the MPU version $\geq 0x06$. For other versions of MPU, this bit is read as zero and ignored on writes. - When this bit is set to 1, the L1 instruction cache is not automatically flushed. Any code in the uncachable region that is already in the L1 instruction cache triggers a cache hit. - When the instruction cache is explicitly disabled by software (<code>IC_CTRL.DC == 1</code>), the code is not cached in any level of the ARC cluster cache hierarchy (L1, L2, and so on). - When a region is fetch protected (that is execute bits <code>KE==0</code> and <code>UE==0</code>), the region must not be marked as uncached (<code>MPU_EN[12]==1</code>).
DC	[13]	Global data cacheability that applies to accesses outside all protected regions. 0x0: Data is cacheable in all levels of the cache hierarchy. 0x1: Data is not cacheable in any level of the cache hierarchy (L1, L2, and so on). Note: This bit is valid only when the MPU version $\geq 0x06$. For other versions of MPU, this bit is read as zero and ignored on writes. The <code>AUX_VOLATILE</code> register programming take precedence over the MPU cacheability attribute.
EN	[30]	MPU enable bit A value of 0 means disabled; 1 means enabled.

11.1.8 MPU Exception Cause Register, MPU_ECR

Address: 0x420

Access: R

Reset: 0x0006_0000

The MPU_ECR register records the memory region that caused the most recent exception and indicates the type of transaction causing the exception.

This register contains 0x00060000 on reset.

The Protection Violation exception, EV_ProtV, is used to signal an MPU violation. This exception is used to also signify a stack out-of-bounds exception or a code protection violation. The EC_CODE field is updated by all of these exceptions, and uses the parameter field (bits 7:0) to indicate the origin of the exception (MPU, Code Protection, Stack out-of-bounds, and so on). The MR field indicates the actual region selected by a faulting address. The special value 0xff for this MR field indicates that no region match was detected.

Figure 11-13 MPU_ECR Register

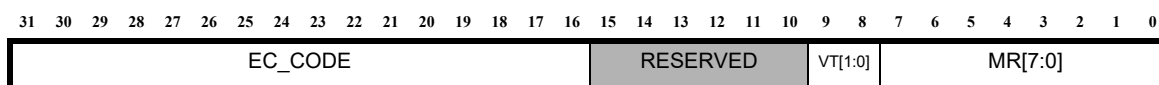


Table 11-7 MPU Exception Cause Register

Field	Bit	Description
MR	[7:0]	MPU Region Indicates which protection region was violated to cause the interrupt according to the following values: ■ 0x00 MPU region 0 ■ 0x01 MPU region 1 ■ 0x02 MPU region 2 ■ 0x03 MPU region 3 ■ 0x04 MPU region 4 ■ 0x05 MPU region 5 ■ 0x06 MPU region 6 ■ 0x07 MPU region 7 ■ 0x08 MPU region 8 ■ 0x09 MPU region 9 ■ 0x0a MPU region 10 ■ 0x0b MPU region 11 ■ 0x0c MPU region 12 ■ 0x0d MPU region 13 ■ 0x0e MPU region 14 ■ 0x0f MPU region 15 0x10 MPU region 16 0x11 MPU region 17 0x12 MPU region 18 0x13 MPU region 19 0x14 MPU region 20 0x15 MPU region 21 0x16 MPU region 22 0x17 MPU region 23 0x18 MPU region 24 0x19 MPU region 25 0x1A MPU region 26 0x1B MPU region 27 0x1C MPU region 28 0x1D MPU region 29 0x1E MPU region 30 0x1F MPU region 31 ■ 0xff Access missed all regions

Table 11-7 MPU Exception Cause Register

Field	Bit	Description
VT	[9:8]	Violation Type Indicates the type of access that caused the violation according to the following values: ■ 0x0: Execution Violation ■ 0x1: Data Read Violation ■ 0x2: Data Write Violation ■ 0x3: Data Read-modify-write Violation (such as EX instruction)
EC_CODE[15:0]	[31:16]	Vector number of the EV_ProtV exception: set to 0x0006

11.1.9 MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB31

Address: Bank 0:
 MPU_RDB0: 0x422, MPU_RDB1: 0x424
 MPU_RDB15: 0x440
 Bank 1:
 MPU_RDB16: 0x422, MPU_RDB17: 0x424
 MPU_RDB31: 0x440

Access: RW

Reset 0x00000000

The MPU region descriptor base registers contain the base address for each region and enables the particular region. These registers are banked. Bank 0 registers are used for setting permissions for regions 0 to 15. Bank 1 registers are used for setting permissions for regions 16 to 31. You can select a bank using MPU_EN[0].

If you attempt to program any of the unavailable MPU_RDB registers (you can configure the number of MPU regions during build-time), an instruction error exception is raised.

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 11-14 MPU Region Descriptor Base Registers

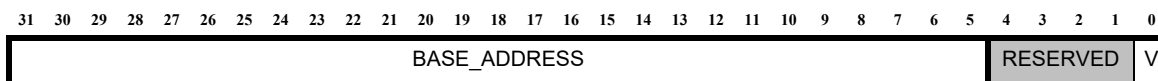


Table 11-8 MPU Region Descriptor Base Registers Field Description

Field	Bit	Description
V	[0]	Region Descriptor Validity Flag ■ 0: This region descriptor is unused. ■ 1: This region descriptor is valid.
BASE_ADDRESS	[31:5]	Base Address Logical base address of the region; must be aligned to a multiple of the region size

11.1.10 MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP31

Address: (Bank 0) MPU_RDP0: 0x423, MPU_RDP1: 0x425 ...
MPU_RDP15: 0x441
(Bank 1) MPU_RDP16: 0x423, MPU_RDP17: 0x425 ...
MPU_RDP31: 0x441

Access: RW

Reset 0x00000000

The MPU region descriptor permissions registers set the size and set the permissions for each region. These registers are banked. Bank 0 registers are used for setting permissions for regions 0 to 15. Bank 1 registers are used for setting permissions for regions 16 to 31. You can select a bank using MPU_EN[0]. This register contains 0x00000000 on reset.

If you attempt to program any of the unavailable MPU_RDP registers (you can configure the number of MPU regions during build-time), an instruction error exception is raised.

Figure 11-15 MPU Region Descriptor Permissions Registers

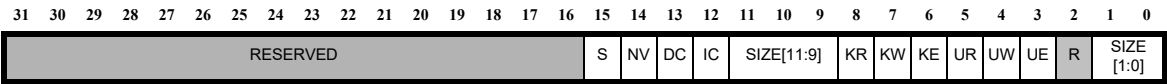


Table 11-9 MPU Region Descriptor Permission Registers Field Description

Field	Bit	Description																														
SIZE	[11:9] and [1:0]	Size; The size of the region is a 5-bit field, the three MSB bits are represented in [11:9] and the two LSB bits are represented in [1:0]. Together these fields specify the size of the region in bytes: <table><tr><td>00000-00011</td><td>Reserved</td></tr><tr><td>0x4 32</td><td>0x5 64</td></tr><tr><td>0x6 128</td><td>0x7 256</td></tr><tr><td>0x8 512</td><td>0x9 1k</td></tr><tr><td>0xA 2K</td><td>0xB 4K</td></tr><tr><td>0xC 8K</td><td>0xD 16K</td></tr><tr><td>0xE 32K</td><td>0xF 64K</td></tr><tr><td>0x10 128K</td><td>0x11 256K</td></tr><tr><td>0x12 512K</td><td>0x13 1M</td></tr><tr><td>0x14 2M</td><td>0x15 4M</td></tr><tr><td>0x16 8M</td><td>0x17 16M</td></tr><tr><td>0x18 32M</td><td>0x19 64M</td></tr><tr><td>0x1A 128M</td><td>0x1B 256M</td></tr><tr><td>0x1C 512M</td><td>0x1D 1G</td></tr><tr><td>0x1E 2G</td><td>0x1F 4G</td></tr></table> Note: The MPU region size has the following constraints: ■ MPU region size >= minimum cache line size of the L1 cache ■ MPU region size >= minimum cache line size of the L2 cache	00000-00011	Reserved	0x4 32	0x5 64	0x6 128	0x7 256	0x8 512	0x9 1k	0xA 2K	0xB 4K	0xC 8K	0xD 16K	0xE 32K	0xF 64K	0x10 128K	0x11 256K	0x12 512K	0x13 1M	0x14 2M	0x15 4M	0x16 8M	0x17 16M	0x18 32M	0x19 64M	0x1A 128M	0x1B 256M	0x1C 512M	0x1D 1G	0x1E 2G	0x1F 4G
00000-00011	Reserved																															
0x4 32	0x5 64																															
0x6 128	0x7 256																															
0x8 512	0x9 1k																															
0xA 2K	0xB 4K																															
0xC 8K	0xD 16K																															
0xE 32K	0xF 64K																															
0x10 128K	0x11 256K																															
0x12 512K	0x13 1M																															
0x14 2M	0x15 4M																															
0x16 8M	0x17 16M																															
0x18 32M	0x19 64M																															
0x1A 128M	0x1B 256M																															
0x1C 512M	0x1D 1G																															
0x1E 2G	0x1F 4G																															
UE	[3]	General Execute Permission ■ 0x0: Instruction fetch is not permitted. ■ 0x1: Instruction fetch is permitted when in user or kernel mode.																														
UW	[4]	General Write Permission ■ 0x0: Region is write-protected. ■ 0x1: Region is writable when in user or kernel mode.																														
UR	[5]	General Read Permission ■ 0x0: Region is read-protected. ■ 0x1: Region is readable when in user or kernel mode. Note that the UR field controls only read access; instruction-fetch access is controlled by the UE field.																														
KE	[6]	Kernel-only Execute Permission ■ 0x0: Instruction fetch is not permitted. ■ 0x1: Instruction fetch is permitted when in kernel mode only.																														
KW	[7]	Kernel-only Write Permission ■ 0x0: Region is write-protected. ■ 0x1: Region is writable when in kernel mode only.																														

Table 11-9 MPU Region Descriptor Permission Registers Field Description

Field	Bit	Description
KR	[8]	Kernel-only Read Permission 0x0: Region is read-protected. 0x1: Region is readable when in kernel mode only. Note that the KR field controls only read access; instruction-fetch access is controlled by the KE field.
IC	[12]	Global code cacheability that applies to this region 0x0: Code is cacheable in all levels of the cache hierarchy. 0x1: Code is not cacheable in any level of the cache hierarchy (L1, L2, and so on). Notes: - This bit is valid only when the MPU version $\geq$ 0x06. For other versions of MPU, this bit is read as zero and ignored on writes. - When this bit is set to 1, the L1 instruction cache is not automatically flushed. Any code in the uncachable region that is already in the L1 instruction cache triggers a cache hit. - When the instruction cache is explicitly disabled by software (IC_CTRL.DC == 1), the code is not cached in any level of the ARC cluster cache hierarchy (L1, L2, and so on). - When a region is fetch protected (that is execute bits KE==0 and UE==0), the region must not be marked as uncached (MPU_RDPn[12==1]).
DC	[13]	Global data cacheability that applies to this region 0x0: Data is cacheable in all levels of the cache hierarchy. 0x1: Data is not cacheable in any level of the cache hierarchy (L1, L2, and so on). Note: This bit is valid only when the MPU version $\geq$ 0x06. For other versions of MPU, this bit is read as zero and ignored on writes. The AUX_VOLATILE register programming take precedence over the MPU cacheability attribute.
NV	[14]	Define this MPU region as non-volatile (NV). 0x0: The memory space for this MPU region is treated as a volatile uncached space. 0x1: The memory space for this MPU region is non-volatile. Note: This bit is valid only when the MPU version $\geq$ 0x07. For other versions of MPU, this bit is read as zero and ignored on writes. The MPU volatile attributes have precedence over the AUX_VOLATILE register programming and MPU cacheability attribute. A peripheral region can also be programmed using this bit.

Table 11-9 MPU Region Descriptor Permission Registers Field Description

Field	Bit	Description
S	[15]	Specifies ordering of transactions in this region when defined as volatile (NV bit 0). 0x0: Relaxed ordering. Reads and writes to this volatile region are not serialized. 0x1: Strict ordering. Reads and writes to this volatile region are serialized by the core. Note: This bit is valid only when the MPU version $\geq$ 0x06. For other versions of MPU, this bit is read as zero and ignored on writes. The MPU strict ordering attribute has precedence over the <code>AUX_VOLATILE</code> register programming.

**Caution**

A region descriptor must not be enabled unless all fields have been set to well-defined values. For example, the region size must be set to a non-reserved value, and all reserved bits must be zero. You must disable the MPU before any region descriptors are updated by setting the `MPU_EN` register to 0x00000000. After all of the region descriptors have been set to well-defined values, the MPU can be re-enabled by setting `MPU_EN` to 0x40000000.

Any MPU region that is programmed with an undefined region size assumes the minimum region size. If the base address is not a multiple of the region size, the effective base address is the highest address that is less than the given address and is also a multiple of the region size. In other words, the base address is the address of the region containing the programmed address.

Table 11-10 Build Configuration Registers

Number	Name	Access	Description
0x6D	Memory Protection Build Configuration Register, MPU_BUILD	R	Build configuration for: memory protection unit

11.1.11 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 11-16 MPU_BUILD Register

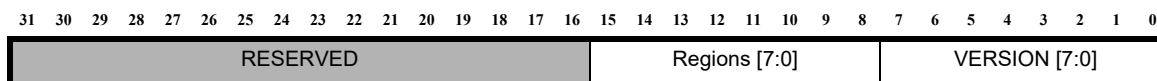


Table 11-11 MPU_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version Indicates the version of the MPU: 0x0 : When no MPU is present 0x3 : MPU with 32 bytes minimum region size 0x6: MPU with 32 regions and support for defining cacheability attributes for each region 0x7: Support for volatile access per MPU region All other values are reserved.
Regions	[15:8]	Number of Regions Indicates the number of regions supported: 0x00 when no MPU is present 0x08 for 8 regions 0x10 for 16 regions 0x12 for 18 regions 0x14 for 20 regions 0x16 for 22 regions 0x18 for 24 regions 0x1A for 26 regions 0x1C for 28 regions 0x1E for 30 regions 0x20 for 32 regions All other values are reserved.

11.1.12 MPU Exceptions

See [Table 5-4](#) on page [233](#).

12

Protection Schemes

ARCV2 ISA-based processors provide several memory-protection options, which are described in this chapter. See the Databook for your specific processor to determine which options are available. These protection options include:

- Stack protection: checks overflow and underflow of reserved stack space
- Code protection: hardware means to block read and write to code space

The different protection schemes may be combined to achieve several levels of protection against malicious or misbehaving code in critical applications.

12.1 Stack Checking

Stack checking is a mechanism for checking stack accesses and raising an exception when a stack overflow or underflow is detected, provided stack exceptions are enabled. The logic triggers an `EV_ProtV` exception when a violation occurs. Topics covered include configuring this option for your core, enabling the option at runtime, and programming the stack range. If stack exceptions are enabled, the memory region in which the stack is located can be explicitly programmed using the following optional auxiliary registers; two for kernel and two for user mode:

- `USTACK_TOP`, 0x260
- `USTACK_BASE`, 0x261
- `KSTACK_TOP`, 0x264
- `KSTACK_BASE`, 0x265

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

12.1.1 Build Configuration Register

[Stack Region Configuration Register](#), `STACK_REGION_BUILD` (0xC5) identifies the presence and version of stack checking in the build.

12.1.2 Enabling Stack Checking

The kernel can enable stack checking by setting the SC bit (bit 14) in the [Status Register, STATUS32](#). In user mode, this bit is not writable and returns zero on read. If stack checking is not configured for the core, this bit is not writable and returns zero on read. In user mode, the STATUS32.SC bit is read as zero and ignored on writes. When stack checking is included in the configuration, the following auxiliary registers are included in the processor:

- KSTACK_BASE, KSTACK_TOP, USTACK_BASE, and USTACK_TOP
- STACK_REGION_BUILD BCR

12.1.3 Specifying Stack Regions

Specify the memory regions reserved for the stack by loading their addresses into the following auxiliary registers. Separate register pairs define the kernel-mode and user-mode stack regions.

**Note**

The stack grows downward from higher memory address to a lower memory address.

- USTACK_TOP (0x260) is the user-mode lower address limit.
- USTACK_BASE (0x261) is the user-mode upper address limit.
- KSTACK_TOP (0x264) is the kernel-mode lower address limit.
- KSTACK_BASE (0x265) is the kernel-mode upper address limit.

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.

12.1.4 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.

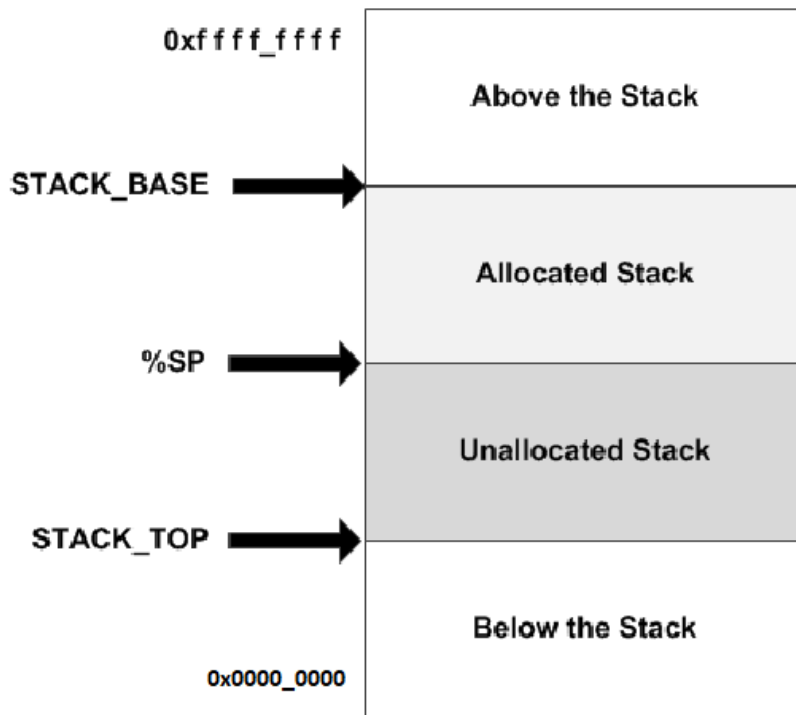
12.1.4.1 Stack Regions

Figure 12-1 shows the stack regions, as defined by the range registers and the architectural stack-pointer register (%SP). The allocated and unallocated regions make up the reserved stack space. The stack grows downward.

- %SP is the architectural stack-pointer register, which usually points to the top of the stack. It is always r28 in ARCV2-based processors.
- The allocated stack region is defined as all addresses from (and including) %SP up to (but excluding) STACK_BASE.
- The unallocated stack region is defined as all addresses from (and including) STACK_TOP up to (but excluding) %SP.
- STACK_BASE points to the first 32-bit word above the reserved stack space, and therefore all addresses greater than or equal to STACK_BASE are above the stack.
- STACK_TOP points to the last allocatable word in the stack region, and therefore all addresses less than STACK_TOP are below the stack.

**Note**

The stack grows downward from higher memory address to a lower memory address.

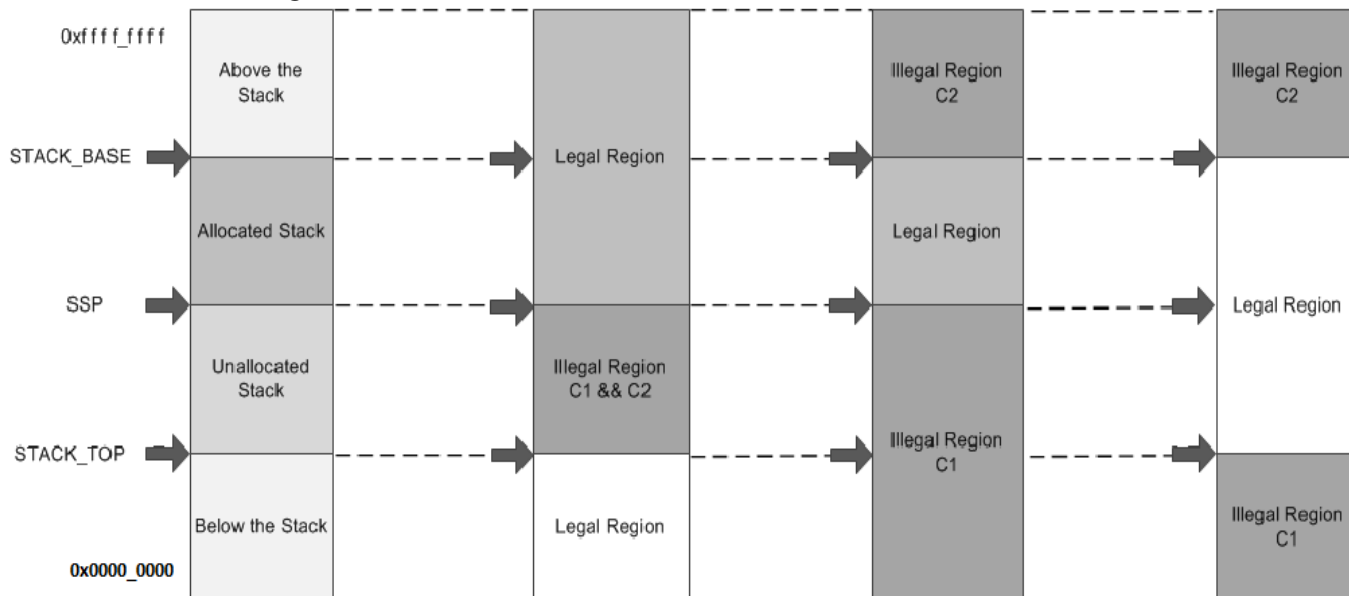
Figure 12-1 Stack regions

12.1.4.2 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 12-2](#) and [Figure 12-2](#) illustrate the conditions checked in three possible scenarios.

Figure 12-2 Stack Checking Scenarios**Table 12-1 Stack Checking Specification**

	Scenario 1 Non-%SP-Based Address.	Scenario 2 %SP-Based, No Auto-Update of %SP	Scenario 3 %SP-Based, with Auto-Update of %SP
Conditions	Example: LD r0, [r1];	Example: LD r0, [r28];	Example: LD.a r0, [r28];
Condition C1	address < SSP	address < SSP	address < STACK_TOP
Condition C2	address ≥ STACK_TOP	address ≥ STACK_BASE	address ≥ STACK_BASE
Condition C3		address < STACK_TOP	
Error Condition	C1 && C2	C1 C2 C3	C1 C2

12.1.5 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 5-4](#) on page [233](#)

12.1.6 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.

12.1.7 Stack Region Auxiliary Register Set

Table 12-2 Stack Region Checking Auxiliary Registers

Auxiliary Register	Name	Description
0x260	USTACK_TOP	User stack top address register
0x261	USTACK_BASE	User stack base address register
0x264	KSTACK_TOP	Kernel stack top address register
0x265	KSTACK_BASE	Kernel stack base address register

12.1.7.1 User Stack Region Top Address, USTACK_TOP

USTACK_TOP Address:	0x260
Access	RW
Reset:	0x0

Figure 12-3 USTACK_TOP when ADDR_SIZE == 32

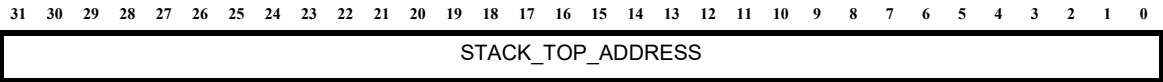
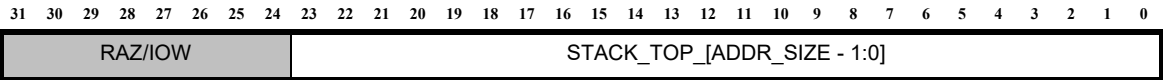


Figure 12-4 USTACK_TOP when ADDR_SIZE < 32



You can use the stack region top address register, USTACK_TOP to set the top address of the valid stack region in memory.

All load and store addresses are checked against the region between USTACK_BASE and USTACK_TOP. For systems with an MMU, the virtual memory addresses are used. Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store is between the SP and USTACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the USTACK_BASE to USTACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

12.1.7.2 User Stack Region Base Address, USTACK_BASE

USTACK_BASE Address: 0x261

Access RW

Reset: 0x0

Figure 12-5 USTACK_BASE when ADDR_SIZE == 32

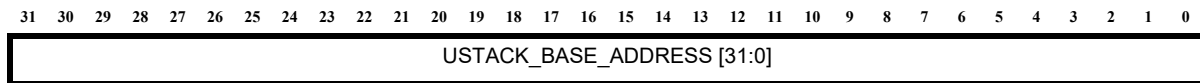
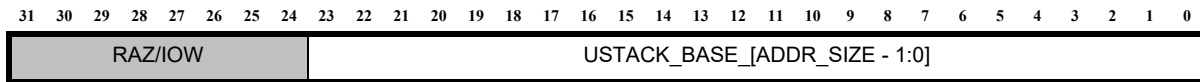


Figure 12-6 USTACK_BASE when ADDR_SIZE < 32



The stack region base address register, USTACK_BASE, is a read or write register, and is used to set the base address of the valid stack region in memory.

You can use the following auxiliary registers to set the stack region in user mode. This auxiliary register is present in the processor if you have configured the processor to include stack checking.



Note

The stack grows downward from higher memory address to a lower memory address.

All load and store addresses are checked against the region between USTACK_BASE and USTACK_TOP. For systems with an MMU, the virtual memory addresses are used.

Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store accesses the region between the SP and USTACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the USTACK_BASE to USTACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

12.1.7.3 **Kernel Stack Region Top Address, KSTACK_TOP**

KSTACK_TOP Address: 0x264

Access RW

Reset: 0x0

Figure 12-7 KSTACK_TOP when ADDR_SIZE == 32

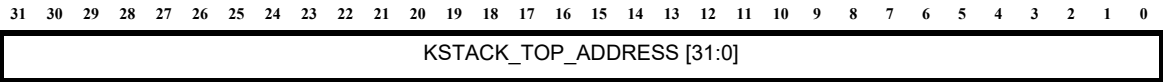
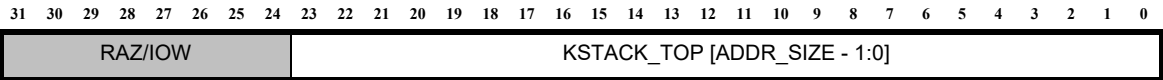


Figure 12-8 KSTACK_TOP, when ADDR_SIZE < 32



You can use the stack region top address register, KSTACK_TOP to set the top address of the valid stack region in memory in kernel mode.

All load and store addresses are checked against the region between KSTACK_BASE and KSTACK_TOP. For systems with an MMU, the virtual memory addresses are used. Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store is between the SP and KSTACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the KSTACK_BASE to KSTACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

12.1.7.4 **Kernel Stack Region Base Address, KSTACK_BASE**

KSTACK_BASE Address:	0x265
Access	RW
Reset:	0x0

Figure 12-9 KSTACK_BASE when ADDR_SIZE == 32

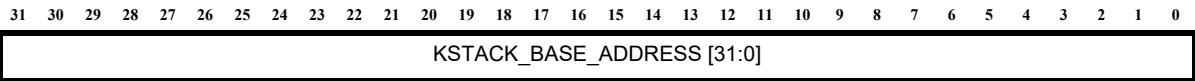
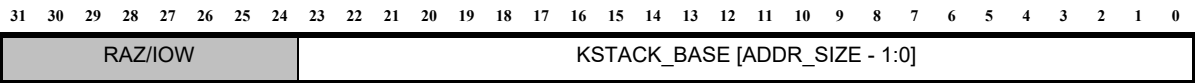


Figure 12-10 KSTACK_BASE when ADDR_SIZE < 32



The stack region base address register, KSTACK_BASE, is a read or write register, and is used to set the base address of the valid kernel stack region in memory.

You can use the following auxiliary registers to set the stack region in kernel mode. This auxiliary register is present in the processor if you have configured the processor to include stack checking.

 **Note** The stack grows downward from higher memory address to a lower memory address.

All load and store addresses are checked against the region between KSTACK_BASE and KSTACK_TOP. For systems with an MMU, the virtual memory addresses are used.

Set the SC bit of STATUS32 to 1 to enable stack checking.

If any load or store accesses the region between the SP and KSTACK_TOP region, a protection violation exception, EV_ProtV, is generated indicating an invalid stack address was accessed.

If a load or store uses SP (PUSH, POP, or LD or ST with SP in the b field) and is outside of the KSTACK_BASE to KSTACK_TOP region, a protection violation exception, EV_ProtV, occurs indicating that an invalid stack pointer (SP) address was used.

12.1.7.5 **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, [User Stack Region Base Address, USTACK_BASE](#) and [User Stack Region Top Address, USTACK_TOP](#).

Figure 12-11 STACK_REGION_BUILD Register

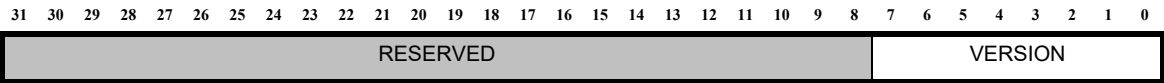


Table 12-3 STACK_REGION_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of stack region checking 0x1: ARC 700 style stack checking 0x2: ARCV2 style stack checking
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

12.1.8 **Stack Checking Exceptions**

See [Table 5-4](#) on page [233](#).

12.2 Coexisting Protection Schemes

Stack Checking can coexist with Code Protection, MPU, or MMU.

- 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
 - MMU permission violations set bit 3

Therefore, multiple protection violations can be reported by raising a single exception. See the [Table 5-4](#) for more information about the exception information for each of the protection schemes.

12.3 Code Protection

This chapter describes the optional code protection mechanism in the ARCV2 processor.

The code protection option is intended to prevent piracy and hacking of code memory by blocking unauthorized loads and stores without impacting processor performance.

12.3.1 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.

12.3.2 Code Protection Behavior

This section describes how the code protection scheme behaves in relation to specific scenarios.

12.3.2.1 Processor Reset

Protection is valid immediately after reset.

12.3.2.2 User and Kernel Mode

The code protection scheme applies to both user and kernel modes.

12.3.2.3 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.

12.3.2.4 DMI Access

DMI access violations do not raise code-protection exceptions. Write commands from DMI are ignored, and read transactions return zero data.

12.3.2.5 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.

12.3.2.6 DFT Scan

Code protection is not available when the core is accessed through the DFT (Design-for-Test) scan chain.

12.3.2.7 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](#).

13

Memory Management Unit

13.1 MMU Introduction

This section describes the ARCV2 Memory Management Unit (MMU).

The Memory Management Unit provides hardware support and acceleration for address translation and protection. In effect the MMU acts as a cache into the page table, using a mechanism known as the *Translation Lookaside Buffer* (TLB). The TLB caches a subset of frequently used page-table entries within the MMU to allow address translation and protection checks to be performed without delays. When a memory location is accessed for which the page table entry is not held in the TLB, the page table must be searched and the appropriate entry loaded, or, if no matching page is found, a page-fault exception is raised.

In some systems the mechanism used to update the set of page table entries held in the TLB is provided by the MMU hardware. Other systems, including the MMU of the ARCV2 family, use a software-managed TLB, where an exception handler updates the TLB entries from the page tables in memory ARCV2.

13.2 Overview of MMU

The following aspects of the ARCV2 Memory Management Unit are covered:

- Memory management options
- Translation lookaside buffer (TLB) structure
- Page descriptor formats
- Memory mapping and operating modes
- Memory management related exceptions
- Writing TLB miss exception handlers

Table 13-1 Special Purpose Registers for TLB Control

Auxiliary Register	Name	Description
0x007	MEMSEG	MEMSEG register (PAE)
0x460	TLBPD0	TLB Page descriptor register 0
0x461	TLBPD1	TLB Page descriptor register 1

Table 13-1 Special Purpose Registers for TLB Control (Continued)

Auxiliary Register	Name	Description
0x463	TLBPD1_HI	TLB Page descriptor register 3
0x464	TLBIndex	TLB index register
0x465	TLBCommand	TLB command register (fully serializing)
0x468	PID	Process ID, TLB enable
0x46A	SASID0	Shared library ASID0
0x46B	SASID1	Shared library ASID1
0x46C	SCRATCH_DATA0	Thirty-two-bit scratch auxiliary register that can be used to store any data. The OS may for example use this register to hold the base address of the first-level page table to speed up page-table access.

These registers may only be accessed in kernel mode. An attempt to access these registers from user mode results in an exception.

In ARC HS family of processors, write operations to the auxiliary registers are generally serializing, that is, a pipeline flush occurs after the auxiliary write operation commits. For best operating-system performance, it is desirable to minimize time spent in TLB miss handlers. For this reason, minimizing pipeline flushes is important.

Writes to auxiliary registers [TLBPD0](#), [TLBPD1](#), [TLBPD0_HI](#), [TLBPD1_HI](#), and [TLBIndex](#) do not affect the operating environment of the processor until the [TLBCommand](#) register is written, so it is possible to make writes to auxiliary registers [TLBPD0](#), [TLBPD1](#), [TLBPD0_HI](#), and [TLBPD1_HI](#) are non-serializing.

Writes to the [TLBCommand](#) or [PID](#) register affect the processor operating environment directly and therefore these writes are serializing. Writes to the [SCRATCH_DATA0](#) register are not serializing. Auxiliary read operations are not serializing.

13.2.1 Upper Physical Memory Address, Register, MEMSEG

Address: 0x07

Access: RG

Reset 0x00

Figure 13-1 MEMSEG Register



When the Physical Address Extension (PAE) module is configured, this register provides the upper address bits for debugger access to processor memory. The debugger-inserted loads and stores bypass the MMU (Memory Management Unit) and the physical address is created by appending MEMSEG register as MSB address bits before being sent to the external buses. Effectively the segment register replaces the MMU translation for the purpose of debug instructions only.

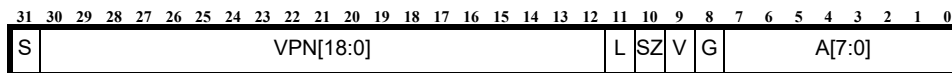
13.2.2 TLB Page Descriptor Register for 32-Bit Virtual Address, TLBPD0

Address: 0x460

Access: RW

Reset 0x00

Figure 13-2 Page Descriptor Format



The MMU supports a fixed 32-bit virtual address space per process but can be configured for a 32-bit or a 40-bit physical address. The page descriptor format is different for these two cases. For a 32-bit physical address, the page descriptor consists of two 32-bit words; the first word relates to TLB Page Descriptor 0 (TLBPD0), the second to TLB Page Descriptor 1 (TLBPD1).

These registers are used for the following purposes:

- To supply a page descriptor for subsequent loading into the TLB (Translation Lookaside Buffer)
- To return a page descriptor from a TLB probe operation
- The virtual page number field is used to specify the virtual address of a TLB entry to be removed (all other fields are ignored).
- On TLB miss exceptions the TLB Page Descriptor register 0, TLBPD0, is updated with the VPN (Virtual Page Number) and ASID (Address Space Identifier) associated with the address that was the cause of the TLB miss exception. To aid the TLB miss handler, the global bit is cleared and the valid bit is set on TLB miss exceptions.

The Reserved fields are read as zero and ignored on writes.

S - Shared Library ASID

This field exists only when `-mmu_shared_lib==true`. Otherwise, this field is read as zero and ignored on writes.

Indicates the entry maps a VA (VPN) that references a shared library and the ASID field indexing bits the SASID registers specifies whether the current processor is subscribed has access to the specific shared library associated with the indexed SASID bit.

VPN[30:M] - Virtual Page Number

This field provides the virtual page number corresponding to this page descriptor. The MMU checks the virtual address of an incoming request against the entries in the translation lookaside buffer. If a matching entry is found, the physical address is calculated. The field is aligned in the page descriptor to allow generation from a 32-bit address using a simple AND operation.

Bit 31 of TLB Page Descriptor 0 is not included in this field because translated memory is only available in the lower two GB of the address space.

The significant virtual page number as a function of the virtual address is calculated as follows:

$$\text{VPN} = \text{VA}[30:M] \text{ where } M \text{ is } \text{Log}_2(\text{pageSize})$$

L – Lock Bit

This field only exists in MMU version 5 or later. When this bit is set to 1, the entry is locked. The LRU algorithm does not consider a locked entry for replacement.

An entry is locked in the TLB at the time of allocation, that is, software must set the lock bit in TLBPD0 (TLBPD0.L) before writing the page in the TLB.

There are two ways to unlock an entry:

- Removing the entry (TLBDeleteEntry) and re-inserting the entry.
- Using the TLBResetLRU command – this unlocks all entries in the Super-page TLB and within a fully associative Normal-page TLB when configured; when a memory-based 4-way NTLB is configured (256 entries or larger), the lock bits are not cleared by the TLBResetLRU command. In this case, software can update the lock bits as necessary using the TLBWrite, TLBInsert, or TLBDelete commands.

Note:

- Entries are only locked in the normal or super-page-size TLBs. Entries in the μ TLBs (ITLB and DTLB) are not locked. The micro-TLBs also do not store the Lock bit and thus it is always read as zero (the bit is RAZ/IOW).
- When a locked entry is invalidated, the lock bit is also cleared.
- The locked entries can be inserted, deleted, or modified using the TLB commands: TLBWrite, TLBInsert, and TLBDelete.

SZ – Page Size

This field exists only when two page sizes are configured, that is, NTLB_NUM_ENTRIES!= 0 and STLB_NUM_ENTRIES!=0.

This field is used to indicate the size of the page mapped by this entry; two page sizes are chosen during processor configuration according to the following ranges, see “[MMU_BUILD](#)”:

- [MMU_BUILD.PSZ0](#): 4k – 16M (normal size page)
- [MMU_BUILD.PSZ1](#): 4k – 16M (super page)

The operating system can define any page to be one of the available sizes. During processor configuration, PSZ1 must be set to be larger than PSZ0 (if both sizes are used). When a single page size is desired, set PSZ0 (MMU_PAGE_SIZE0) to the desired size and configure STLB size to be 0 entries (and insure all PTEs specify page size 0). When an instruction cache or data cache are present, the tag matching logic is configured to handle aliasing for the smallest configured MMU page size.

V - Valid

This bitfield is used to indicate whether an entry in the TLB should be considered during a memory access when checking for a matching TLB entry. When cleared, it indicates that the entry slot is vacant and may be picked for replacement - invalid entries are always chosen before a valid entry is evicted.

G - Global

This bit is used to indicate whether the page is global:

- When set true (1)
 - This page appears in all virtual address spaces. ASID bits for this entry are ignored.
- When set false (0)
 - And if the Shared (S) bit is 1, this page appears in a shared library as described by the SASID registers (and ASID bits A[5:0]).
 - And if the Shared (S) bit is 0, this page appears in a single virtual address space as described by the ASID bits A[7:0].
- If the page is to be available in all address spaces, with identical access requirements, a single page table entry may be used with the global bit set.
 - It is the responsibility of the operating system to ensure that a globally available page does not overlap with a page at the same location in a single address space. This condition causes a fatal machine-check exception.
 - It is possible to set up a global page accessible only by the operating system in kernel mode, and a page at the same virtual address only accessible by one user mode task. However, page read/write/execute permissions are not considered when testing for multiple overlapping. Do not attempt to set up a global page accessible only by the operating system in kernel mode, and a page at the same address that is only accessible by one user-mode task; this arrangement is not permitted and causes a fatal machine-check exception.
- If the page is to be available in some but not all address spaces, or if it is to be available in all address spaces, but with different access privileges, a separate page-table entry is required for each virtual address space in use.

The global bit is typically used for mapping private operating system memory pages, in which case it is not used with user mode read/write/execute permissions.

In some operating systems, this mode may be used for data or code areas shared between all processes and the operating system.

A[7:0] - Address Space Identifier ASID

These bits describe the eight-bit address space identifier (ASID) that can be considered to be an extension of the virtual address.

When the global bit G is set false, shared-library ASID support is enabled, and a memory access is taking place, ASID[5:0] is compared against the corresponding bit position in the [SASID0](#) [SASID1](#) registers to determine whether a TLB match has taken place. If the indexed bit is set, the entry is a shared library hit.

If shared library support is disabled, or there is no shared library ASID hit, and the global bit G is set to false, and a memory access is taking place, the ASID bits are tested against the current task's ASID from the machine status register (PID) to determine whether a TLB match has taken place. With the eight-bit ASID, the processor supports up to 256 concurrent virtual address spaces.

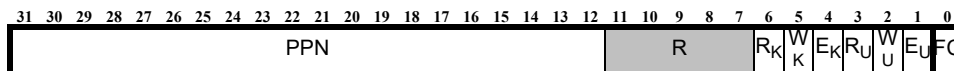
13.2.3 TLB Page Descriptor Register for 32-Bit Physical Address, TLBPD1

Address: 0x461

Access: RW

Reset 0x00

Figure 13-3 TLB Page Descriptor Register, TLBPD1



The MMU supports a fixed 32-bit virtual address space per process but can be configured for a 32-bit or a 40-bit physical address. The page descriptor format is different for these two cases. For a 32-bit physical address, the page descriptor consists of two 32-bit words; the first word relates to TLB Page Descriptor 0 (TLBPD0), the second to TLB Page Descriptor 1 (TLBPD1).

The Reserved fields are read as zero and ignored on writes.

PPN - Physical Page Number

This field is used for virtual-to-physical address translation, as follows:

- When the entry is marked valid:
 - This virtual memory page is present in physical memory.
 - This field describes the physical page in the main memory that is used for accesses to the virtual page. The high order bits of the physical address (page number) come from this field, and the bottom twelve bits (page offset) come from the bottom twelve bits of the virtual address.
 - The physical page can be located anywhere in the full 32-bit (4 GB) address space or 40-bit (1 TB) address space.
- When the entry is not marked valid:
 - This virtual page is not present in physical memory. This arrangement is only useful in an operating system supporting demand-paged virtual memory.
 - This field is typically used by the OS to indicate the location of the page in a disk-based swap file.

The physical page number of a block is the address of the block shifted right by $\log_2(\text{PageSize})$ bits (divided by *PageSize*).

R_K, W_K, E_K – Kernel Mode Permission Bits

Each page descriptor features separate access control bits for user-mode and kernel-mode tasks.

These bits do not form part of the addressing mechanism. It is not permissible to have more than one TLB entry mapped to the same virtual address, even if the access permissions do not overlap.

An exception is generated if an access is attempted that violates the access permissions for the page. The access does not complete.

These three bits control the permissions granted to tasks, interrupts, or exception handlers running in kernel mode or other operating-system functions using kernel mode. Setting a bit to true (1) indicates that the permission is granted.

- R_K — Kernel mode read permission
- W_K — Kernel mode write permission
- E_K — Kernel mode execute permission

R_U, W_U, E_U — User Mode Permission Bits

Each page descriptor features separate access control bits for user-mode and kernel-mode tasks.

Note that these bits do not form part of the addressing mechanism. It is not permissible to have more than one TLB entry mapped to the same virtual address, even if the access permissions do not overlap.

An exception is generated if an access is attempted which violates the access permissions for the page. The access does not complete.

These three bits control the permissions granted to tasks running in user mode. Setting the bit true (1) indicates that the permission is granted.

- R_U — User mode read permission
- W_U — User mode write permission
- E_U — User mode execute permission

F_C - Cached/Uncached Flag

This bit controls the data cache operation when accessing this page of system memory.

- The default is to set this bit to true to indicate that caches may be used for accesses to this page.
- When this bit is set to false, accesses to this page are sent directly to external memory, bypassing caches. This setting can be used for I/O register space or volatile data areas, such as a region of memory where data is written by a DMA transfer.

It is the responsibility of the operating system (or user code) to ensure that the cache does not contain entries from pages that are marked uncached. This requirement ensures that the cache does not contain old data that might either be flushed into the uncached page at some later point, or that might be used incorrectly if a page is subsequently marked cached.

When more than one virtual page is mapped to the same physical page, all pages must have the same setting for the cached/uncached flag.

In addition to the cached/uncached switch in the page descriptor, individual load and store instructions have a cached/uncached mode switch. Accesses are performed without caches if either the instruction or the page descriptor indicates uncached. Accesses use the caches only if both the instruction and the page descriptor indicate cached.

For data items that span a page boundary and the two relevant page descriptors' F_C flags are different, an exception is generated indicating data access spanning multiple memory targets. However, if such an access uses the `.di` or direct memory (cache bypass) switch, then the memory access of both data fragments are considered uncached and thus do not result in the multiple memory targets exception.

In addition to the page descriptor FC bit and the `.di` instruction switch, there are other mechanisms that render a data access uncached. These include the `DMP_PER_AUX` region and `AUX_VOLATILE` parameters; any of these attributes individually can render the data access uncached.

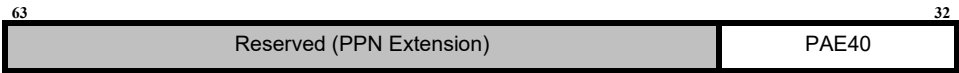
13.2.4 TLB Page Descriptor Register for 64-Bit Physical Address, TLBPD1_HI

Address: 0x463

Access: RW

Reset 0x00

Figure 13-4 TLBPD1_HI Register



For a 40-bit physical address, this register holds the most significant 8 bits of the physical address. When the physical address extension is not enabled, this register is read as zero and ignored on writes.

13.2.5 TLB Index Register, TLBIndex

Address: 0x464

Access: RW

Reset 0x00

Figure 13-5 TLBIndex Register



This register is set by the programmer to communicate the index for TLBWrite and TLBRead commands, and set by the hardware to communicate a result from the TLBGetIndex and TLBProbe commands. Bit 31 is set to indicate an error: A value of 0x8000.0000 or above indicates an error. Writes to this register can be non-serializing. The address in the TLBIndex register is mapped as shown in [Table 13-2](#).

The Reserved field is set to zero.

The following fields are described in more detail:

- [Index](#)
- [RC](#)
- [E, Error Code](#)

Index

This field is read/write.

This part of the register is set by the programmer to communicate the index for the TLBWrite and TLBRead commands, and set by the hardware to communicate a result from the TLBINSERT/TLBDELETE, TLBGetIndex, and TLBProbe commands. If an error has occurred (E is set), the index contains the error code. See [Table 13-4: “TLB Command Register Command List”](#) for more information on usage.

Table 13-2 TLBIndex Addresses

Access Type	Address	Description
NTLB	0x0-0x7FF (max)	This field allows both TLBWrite and TLBRead to be performed on the NTLB RAM. VERSION 4 MMU TLB depth up to 2K
STLB	0x800-0x80F (max)	Supports both TLBWrite and TLBRead to be performed on the STLB RAM.
μTLB	0x1000	Supports TLBRead to be performed on the μTLB.
μDTLB	0x1100-	Supports TLBRead to be performed on the μDTLB.

RC

This field is read-only. This three-bitfield is set by the hardware indicating results from certain commands. Writes to this field are ignored.

Table 13-3 Result Code, Read Only

Code	Description
0x0	Normal operation
0x2	Incompatible size and index detected
0x4	Duplicate TLB entries
0x7	Uncorrectable ECC error in the NTLB RAMs

E, Error Code

This field is read-only.

This bit is set by the hardware when an error has occurred or a searched entry is not found. Writes to this bit are ignored.

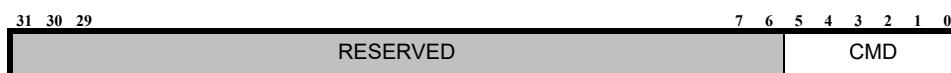
13.2.6 TLB Command Register, TLBCommand

Address: 0x465

Access: W

Reset: 0x00

Figure 13-6 TLBCommand Register



This fully serializing register is used to initiate all transactions with the TLB. Data is communicated through the `TLBPD0`, `TLBPD1`, and `TLBIndex` registers. If an invalid command is written, a machine check exception is raised. Only bits [5:0] are considered in decoding command and detecting invalid command. For invalid commands the Instruction Error ECR value is 0x020000. The TLB command operations can still be performed when the MMU is disabled (when the T bit is 0 in the [PID](#) register).

The following commands are supported:

Table 13-4 TLB Command Register Command List

Command	Name	Description
0x1	TLBWrite	Write a TLB entry to the index location specified in <code>TLBIndex</code> and invalidate μ TLBs. This command is also used to remove entries. You can invalidate μ TLB entries by clearing their valid bits, but the entries are otherwise unchanged.
0x2	TLBRead	Read a TLB entry into <code>TLBPD0</code> or <code>TLBPD1</code> from the location specified in <code>TLBIndex</code> .
0x3	TLBGetIndex	Set <code>TLBIndex</code> to contain a suitable index location for the page descriptor in <code>TLBPD0</code> or <code>TLBPD1</code> or an error code. The <code>TLBGetIndex</code> command always returns an index where the new entry can be loaded. The <code>TLBIndex.E</code> bit and <code>RC</code> field are also updated after execution of <code>TLBGetIndex</code> and only the TLB section (NTLB or STLB) for the given size is searched for a suitable index.
0x4	TLBProbe	Searches the JTLB and if specified entry is present in nTLB or sTLB, returns the index of the matching entry in the <code>TLBIndex</code> register. The <code>E</code> and <code>RC</code> fields of <code>TLBIndex</code> are also updated. The page descriptor bits <code>TLBPD0.SZ</code> , <code>V</code> , <code>G</code> and shared bits are not considered in the search.
0x5	TLBWriteNI	Write a TLB entry to the index location specified in <code>TLBIndex</code> , without invalidating μ TLBs. Used for update on TLB miss. Present from version 0x02 on.

Table 13-4 TLB Command Register Command List (Continued)

Command	Name	Description
0x6	IVUTLB	Invalidate μ TLBs. Used when removing entries from the page table in memory. Entries are invalidated by clearing their valid bits but the entries are otherwise unchanged. Present from version 0x02 on.
0x7 0xA	TLBInsertEntry TLBInsertEntryJ	Write a TLB entry to the JTLB index location appropriate for the given VPN in TLBPD0 without invalidating μ TLBs. The TLBIndex register does not need to be set before issuing this command; instead, the command returns the location where the hardware placed the new entry in JTLB in TLBIndex. Note that all matching entries in all TLB sections are deleted before the new entry is inserted. The TLBPD0.V (valid) bit must be set for any entries to be considered a match. If TLBPD0.V bit is 0, then no insertion or deletion is done and the TLBIndex fields E, RC, Index are returned as 1,0x0,0x0. The matching algorithm of Insert is identical to that of delete commands and thus, delete is a subset function of insert. TLBPD0: Size, Global and Shared bits are also inputs to the insert commands; they affect the scope of search in finding entries to delete first. If the given entry TLBPD0.Size bit is set (super page), then the new entry is placed in the STLB; otherwise, for a normal page, it is placed in the NTLB.
0x8	TLBDeleteEntry	Invalidate entry for the page descriptor in TLBPD0, if present in the TLB (including all TLB caches: NTLB, STLB, uTLB, uDTLB). The TLBPD0.V (valid) bit must be set for any entries to be considered a match (and deleted). If TLBPD0.V bit is 0, then no insertion or deletion is done and the TLBIndex fields E, RC, Index are returned as 1,0x0,0x0. TLBPD0: Size, Global and Shared bits are also inputs to the delete commands; they affect the scope of search in finding entries to delete.
0x9	TLBDeleteEntryIS	Identical to TLBDeleteEntry except that in finding entries to delete, a shared entry does not consider the shared library subscriptions ("IS" suffix = 'Ignore Subscriptions'). Thus, a valid shared TLB entry matching the TLBPD0 VPN is invalidated regardless of the state of the relevant SASID bit. However, the global shared ASID enable bit (PID.S) must be set for a shared TLB entry to match in the search for entries to be deleted. This command is available only if -mmu_shared_lib==true.

Table 13-4 TLB Command Register Command List (Continued)

Command	Name	Description
0xB	TLBResetLRU	This command resets the LRU state and unlocks all entries in the Super-page TLB and within a fully associative Normal-page TLB when configured; when a memory-based 4-way NTLB is configured (256 entries or larger), the lock bits are not cleared by the TLBResetLRU command. In this case, software can update the lock bits as necessary using the TLBWrite, TLBInsert, or TLBDelete commands. After reset the LRU state is entry <i>N-1</i>, entry 0 is the LRU and entry 0 is the Most Recently Used (MRU), where <i>N</i> is the maximum number of allocatable entries.

TLBWrite and TLBWriteNI Commands

These commands are used to load an entry into the TLB at the specified index location, and optionally clear (invalidate) the μ TLBs. When a new entry is successfully created in the TLB using TLBWrite, this new entry is promoted to MRU.

TLBWrite is available in all MMU versions, and always invalidates μ TLBs.

TLBWriteNI is available from MMU version 0x02, and does not invalidate μ TLBs.

The operating system may determine an appropriate location for the entry by itself, or may ask the MMU hardware for a suggestion by using the TLBGetIndex command.

The TLBWrite command is also used to remove (shoot down) existing entries, by loading an entry with the V bit set false. The TLBPD0 and TLBPD1 register bits are typically set to all zeros before a TLBWrite command is issued. The operating system may determine the index of the entry to be removed, or may use the TLBProbe command to return an index that corresponds to a virtual address/ASID combination.

The TLBWrite and TLBWriteNI commands can still be performed when the MMU is disabled (when the T bit is 0 in the PID register).

TLBWrite and TLBWriteNI usage:

- Page descriptor to be loaded into the TLB is brought into the TLBPD0 and TLBPD1 auxiliary registers.
- TLBIndex contains the index location to which the entry is to be loaded.
- TLBWrite[Ni] is command code is written to the TLBCommand register.
- TLBPD0 and TLBPD1 auxiliary registers are unchanged after the TLBWrite and TLBWriteNI operations.
- If an invalid index value (out of range) is supplied, the TLB load request is ignored, and TLBIndex is loaded with error flag E set, and the result code field containing error code 0x0 (full value returned is 0x8000.0000). Note the μ -TLBs (uITLB and uDTLB) are not writable and the index value for them is considered 'out of range' for TLBWrite[Ni] operations.
- If an entry with incompatible page size and index is supplied (normal page with index targeting STLB or super page with index targeting NTLB) the TLB load request is ignored, and TLBIndex is loaded with error flag E set and the result code field containing error code 0x2.

- Invalid entries may be loaded ($V=0$). Such entries are not considered during lookup operations. However, this feature allows an entry to be invalidated and also allows a save or restore of the entire main TLB contents to be performed.
- The TLBWrite command may insert, delete, or modify locked entries. The lock and valid bits are treated independently by the TLBWrite command. If the valid bit in PD0 (PD0.V) bit is zero and the lock bit in PD0 (PD0.L) is one, the matching entry is invalidated but its lock bit is set.

TLBRead Command

This command is used to read an entry from the TLB, at the specified index location. The operating system may either determine the location to be read, or may use the TLBProbe command to obtain the location of an entry from a virtual address.

The TLBRead command operation can still be performed when the MMU is disabled (when the T bit is 0 in the PID register).

TLBRead usage:

- The TLBIndex register contains the location from which the entry is to be read.
- The read and write permission bits (in total four bits) are always set to zeros when reading entries in the μ ITLB. Read and write permissions only apply to the μ DTLB.
- The execution permission bits (two bits) are always set to zeros when entries in the μ DTLB are read. Execution permissions only apply to the μ ITLB.
- The reserved bits are always set to zeros when entries in the Joint TLB are read (writes to these bits are ignored).
- TLBPD0 and TLBPD1 registers contain the TLB entry from the specified location. Entries in the TLB that are marked as invalid are returned as they appear in the TLB.
 - When a valid index is used for the read, the TLBIndex register Error flag (E) and the Result Code (RC) field is cleared.
 - If an invalid index value (out of range) is supplied, the TLBIndex is loaded with error flag E set and the index field containing error code 0x0 (full value returned is 0x8000.0000), and the TLB read operation returns an entry with all bits set to zero.

TLBGetIndex Command

This command is used to allow the hardware to provide a TLB index to which a new entry may be loaded. This mechanism has a number of benefits:

- The mechanism enables the creation of simple and fast TLB miss handlers that are independent of the size and associativity of the underlying TLB, and can rely on the hardware to manage the replacement algorithm.
- An operating system that is aware of the configuration of the TLB can implement a different or more sophisticated replacement algorithm than is supported by the hardware — at the cost of increasing the number of cycles taken during TLB misses.
- The handling of complex error conditions may be deferred to the operating system.

The TLBGetIndex command operation can still be performed when the MMU is disabled (when the T bit is 0 in the PID register).

TLBGetIndex usage:

- The page descriptor to be loaded into the TLB is brought into the TLBPD0 and TLBPD1 special-purpose registers. The page descriptor to be loaded into the TLB is written by software into the TLBPD0 and TLBPD1 registers. The page size bit (PD0.SZ) signifies whether the new entry is a normal page or a super page. This bit is used by the hardware to determine which JTLB section, NTLB or STLB, is searched for a suitable location for the new entry. If the size bit is set indicating super page, the index returned is in STLB; if cleared, the index returned is in NTLB.
 - On executing TLBGetIndex, the TLBIndex register is loaded with the location to which the supplied page descriptor can be loaded. The replacement algorithm is true-LRU (binary tree) in the STLB. The NTLB implements a true-LRU replacement policy algorithm when the number of entries is less than 256. The NTLB implements a pseudo-random replacement policy algorithm when the number of entries is greater than 256. When a LRU selected entry is locked, the next 'most-recently-used', unlocked entry is selected.
 - When there is no uncorrectable ECC error in the NTLB PD0 RAM (if accessing NTLB), or always when accessing STLB, the E and RC bits are cleared on execution of TLBGetIndex, and a suitable index for the new entry is always returned in the TLBIndex.Index field.
 - If an uncorrectable ECC error is detected during TLBGetIndex (or any command which reads the NTLB), the E and RC bits are set to 1, 0x7 respectively. The index returned reflects that any cache ways with uncorrectable error behave as if their Valid bit was cleared; thus, like other invalid ways, they are preferred in selection as victims among the four ways; the lowest index invalid way is selected. A suitable index value for the new entry is always returned.
- Invalid ways are selected first, before a true-LRU generated victim is considered.
 - If a getIndex command encounters an uncorrectable ECC error in the NTLB PD0 RAM (while accessing the Valid bits of NTLB entries), those cache ways with uncorrectable error behave as if their Valid bit was cleared.
- This command fails when all entries are locked, that is, the error bit (TLBIndex.E) is asserted and the reason code (TLBIndex.RC) is set to zero.
- This command always returns the LRU entry in the Index (TLBIndex.Index) field.

TLBProbe Command

Searches the JTLB for an entry specified by the PD0 register. If the entry is present in nTLB or sTLB, this command returns the index of the matching entry in the TLBIndex register. If the entry is not found, the TLBIndex register E flag is set along with Result Code 0, and the index field returns 0.

This command searches both the NTLB and STLB for the given VA/ASID. The TLBProbe command operation can still be performed when MMU is disabled (when the T bit is 0 in the PID register).

TLBProbe usage:

- The VPN[VPN_MSB:0] field of TLBPD0 contains the virtual address for which the main TLB is to be searched. The A[7:0] field contains the address space identifier (ASID) to be used for the search (if not global or shared). The page is descriptor bits TLBPD0.Size, V, G and Shared are not considered in the search.

- ❑ As a result of the command, if a single matching entry is found the `TLBIndex` register is loaded with the index of the matching valid entry and the error flag and result code fields are both set to 0. Both NTLB and STLB are searched.
 - ❑ If no matching entry is found in the main TLB, the `TLBIndex` is loaded with error flag `E` set and the Result Code field containing error code 0x0, and the `TLBIndex . Index` field 00x0.
 - ❑ If more than one matching entry is found in the main TLB, the `TLBIndex` register is loaded with error flag `E` set and the Result Code field containing error code 0x4.
 - ❑ It is possible that during the probe, both NTLB and STLB match indicating overlapping entries (normal page overlaps super page). This can only happen if the software manually writes entries incorrectly (overlapping) into these TLBs; that is with matching upper bits of VA and matching ASIDs. This cannot happen if the `TLBGetIndex`, `TLBProbe`, or `TLBInsertEntry` commands are used as they consider size when choosing an index for insertion as well as detecting existing entries that may overlap (match).
 - ❑ If an uncorrectable ECC error is detected during `TLBProbe` (or any command which reads the NTLB), the `E` and `RC` bits are set to 1, 0x7 respectively. The index returned reflects that any cache ways with uncorrectable error behave as if their `Valid` bit was cleared. Thus, if the probe hit two entries in the NTLB but one of those entries had an uncorrectable error detected, then only the other (error free) entry is declared a match; it's index is returned but the `E/RC` code returned is 1, 0x7.
- A matching entry for the `TLBProbe` command is defined as a TLB entry for which:
 - ❑ The TLB entry valid (`V`) bit is set true, and
 - ❑ Significant bits of the virtual address `VPN` field specific to the page size configured for each JTLB section, match the requested (`PD0`) `VPN` exactly, and
 - ❑ Either the TLB entry global (`G`) bit is set, or, if the TLB entry shared (`S`) bit is set then the `SASID` bit indexed by the entry's `ASID[5:0]` is set or, if the `S` bit is clear then the `ASID` field `A[7:0]` matches requested (`PD0`) `ASID` exactly. No other information is used for matching - User/Kernel mode permissions and flag bits are not considered. Note that the `G` bit has priority over the `S` bit and may result in matches with any `S` or `ASID` value.
 - ❑ If a probe command encounters an uncorrectable ECC error in the NTLB `PD0` RAM (while accessing the `Valid` bits of NTLB entries), those cache ways with uncorrectable error behave as if their `Valid` bit was cleared.
 - Note that in version 0x02 and later of the MMU, it is possible for μ TLBs to contain entries that are not present in the main TLB. Be aware that the `TLBProbe` command only searches entries in the main TLB, and not the μ TLBs. `TLBDeleteEntry` can be used to delete a specific entry from all TLB sections. The `TLBRead` command can be used to examine the μ TLB entries one-by-one, or the `IVUTLB` command can be used to explicitly clear the μ TLBs when necessary.

IVUTLB Command

Available from MMU version 0x02 on.

This command is used to allow the operating system to explicitly clear the μ TLBs when the page table in memory is changed. This command invalidates all entries in the μ TLBs by clearing the valid bits of entries (in `PD0`) but does not otherwise change the entry data. Typically when performing page table maintenance operations, the OS uses `TLBDeleteEntry` to remove a specific entry from all TLB sections. In version 0x02

or later of the MMU, when `TLBWriteNI` or `TLBInsertEntry` is used by TLB miss handlers, it is possible for a page table entry to be present in one or both μ TLBs but not present in the main TLB. The `TLBProbe` command does not locate the entry as it is not in the main TLB.

No parameters are required.

TLBInsertEntry, TLBInsertEntryJ Commands

The insert commands are used to add a new entry to the MMU TLBs; the insert commands delete the matching TLB entries in any of the TLB caches including the NTLB, STLB, μ ITLB and μ DTLB. The search for entry deletion spans all TLB caches (including the μ TLBs) regardless of the presence of the entry in the JTLB. The insert commands consider the `TLBPD0` page attribute bits (Shared, Size, Valid, Global) in determining the scope of the search for entries to delete and the TLB where the new entry is inserted. The `TLBPD0.V` (valid) bit must be set for any entries to be considered a match. If `TLBPD0.V` bit is 0, then no insertion or deletion is done and the `TLBIndex` fields are returned as `E==1`, `RC==0x0`, `Index=0x0`. The matching algorithm of Insert is identical to that of delete commands and thus, delete is a subset function of insert. When a new entry is successfully created in the TLB using `TLBInsert`, this new entry is promoted to MRU.

When inserting a super-page, the search within STLB, and μ TLBs for matching entries to be deleted first, before inserting the new page, matches TLB entries VPN in the super-page range of the new entry (even if the TLB entry is a normal page); this helps avoid creating overlap with existing TLB entries. However, Note that NTLB search does not span a full super-page and deletes only the normal page referenced by `PD0.VPN`.

Similarly, when the new entry is a global entry (`TLBPD0.G=1`), the search for matching entries to delete doesn't consider the shared bit or ASID fields of TLB entries.

A shared TLB entry matching the `TLBPD0.VPN` is invalidated only if the relevant `sasid` bit, the bit referenced by the TLB entry's ASID field, is set. Also, the global shared-ASID enable bit (`PID.S`) must be set for a shared TLB entry to match in the search for entries to be deleted. The search would have resulted in a match during a normal instruction or data lookup. Normal (non-shared, non-global) entries cannot match a shared entry being inserted (as the `TLBPD0.ASID` field is a SASID index not a PID).

`TLBInsertEntry`: Used for update on TLB miss.

The new entry is also forwarded to exactly one of the μ TLBs where it replaces a matching entry or an invalid entry or the LRU entry. Which μ TLB the new entry is forwarded to, depends on the execute bits; if either one or both of the execute bits is set, the entry is forwarded to μ ITLB, otherwise, to the μ DTLB. The μ TLB which is not updated with the new entry however gets a request to invalidate any matching entry to avoid stale entries with the same VPN/asid or VPN/sasid subscription.

`TLBInsertEntryJ`: Used when many new entries are to be added to TLB. The new entry is inserted only in the JTLB and not forwarded to either μ TLB.

Both `TLBInsertEntry` and `TLBInsertEntryJ`: If the new inserted entry did not match an existing entry in any TLB cache, the `TLBIndex.E` flag is set and the `RC` field is cleared to 0. If any single TLB Cache had multiple matching entries or NTLB and STLB both had one or more matches, the `TLBIndex.E` flag is set and the `RC` is set to 0x4 and the `Index` field is set to 0, the multiple matching stale entries are automatically invalidated by the hardware as described in the `TLBDeleteEntry` command. The newly inserted entry retains the valid bit that was present in `PD0` during the `TLBInsertEntry` command execution. Present from version 0x04 on.

If an uncorrectable ECC error is detected during insert (or any command which reads the NTLB), the `E` and `RC` bits are set to 1, 0x7 respectively. The index returned reflects that any cache ways with uncorrectable

error behave as if their Valid bit was cleared. Thus, if the insert hit two entries in the NTLB but one of those entries had an uncorrectable error detected, then only the other (error free) entry is declared a match; it is replaced with the new entry and its index is returned in `TLBIndex`, but the E/RC code returned is 1, 0x7.

TLBDeleteEntry, TLBDeleteEntryIS Commands

The delete command is used to invalidate specific entries from the MMU TLB caches. The command deletes matching entries in all four TLB caches including the NTLB, STLb, μ ITLB and μ DTLB. The search for entry deletion spans all TLB caches regardless of the presence of the entry in the JTLB. The delete commands consider the `TLBPD0` page attribute bits (Shared, Size, Valid, Global) in determining the scope of the search for entries to delete. The `TLBPD0.V` (valid) bit must be set for any entries to be considered a match (and deleted). The matching algorithm of delete operations is identical to that of insert; see the insert command for details on the entries matched and deleted.

If the delete command does not match (and delete) any existing entry in any TLB cache, the `TLBIndex.E` flag is set and the RC and Index fields are cleared to 0.

When a delete command finds a single matching entry in JTLB (either NTLB or STLb), and the μ TLBs did not report multiple match errors, the `TLBIndex` register is updated with the index of the matched (and deleted) entry. The `TLBIndex.E` and RC fields are cleared to 0. This is the normal case of entry deletion.

If either NTLB or STLb had multiple matching entries, or NTLB and STLb both had one or more matches, the `TLBIndex.E` flag is set and the RC is set to 0x4. In this case, all of the multiple matching entries are automatically invalidated by the hardware.

Entries in STLb and the μ TLBs are invalidated by clearing their valid bits but the entries are otherwise unchanged. Entries in the RAM based NTLB are invalidated by clearing their valid bits and copying all other fields of `TLBPD0` to the matched entries being deleted. The PD1 descriptor of (matched) TLB entries is not modified for any TLB.

If an uncorrectable ECC error is detected during delete (or any command which reads the NTLB), the E and RC bits are set to 1, 0x7 respectively. The index returned reflects that any cache ways with uncorrectable error behave as if their Valid bit was cleared. Thus, if the delete hit two entries in the NTLB but one of those entries had an uncorrectable error detected, then only the other (error free) entry is declared a match; it is deleted and its index is returned in `TLBIndex`, but the E/RC code returned is 1, 0x7.

TLBResetLRU

This command resets the LRU state and unlocks all entries in the Super-page TLB and within a fully associative Normal-page TLB when configured; when a memory-based 4-way NTLB is configured (256 entries or larger), the lock bits are not cleared by the `TLBResetLRU` command. In this case, software can update the lock bits as necessary using the `TLBWrite`, `TLBInsert`, or `TLBDelete` commands.

After reset the LRU state is entry $N-1$, entry 0 is the LRU and entry 0 is the Most Recently Used (MRU), where N is the maximum number of allocatable entries.

13.2.7 Process Identity Register, PID

Address: 0x468

Access: RW

Reset 0x00

Figure 13-7 Process Identity Register, PID Register Description



The Process Identity register (PID) contains privilege bits that control permissions that can be optionally extended to a user mode task, an address space identifier (ASID) field used by the memory management system, and compatibility mode bits. This is a fully serializing register.

The Reserved fields are read as zero and ignored on writes. The following fields are described in more detail:

- [T, Global TLB Enable](#)
- [S, Shared Library ASID Enable](#)
- [P\[7:0\], Address Space Identifier ASID](#)

T, Global TLB Enable

The Global TLB Enable bit is used to enable or disable the MMU. When this bit is set to 0 the MMU is disabled, which means that all logical addresses are mapped directly to physical addresses. The MMU needs to be enabled (Global TLB Enable bit set to 1) for memory protection and cacheability to work on individual pages. This field is set to 0x0 on reset.



Note

A machine check exception causes the Global TLB enable to be cleared (set to zero).

S, Shared Library ASID Enable

This bit is used to enable the shared library ASID matching in the MMU. If this bit is not set, any entry in the MMU that has its S bit set cannot be detected as a match. Within a TLB entry the PD0.G (Global) bit has priority over the PD0.S (Shared) bit so that an entry may match when G bit is set, even if S bit is not set. This bit is available only when `-mmu_shared_lib==true`.

P[7:0], Address Space Identifier ASID

The eight-bit Address Space Identifier (ASID) is set by the operating system as the ASID of the currently executing process. The ASID is used by the operating system and memory management hardware to allow physical pages to be mapped into many separate virtual address spaces. This field is set to 0x0 on reset.

Typically each independent task has its own ASID value. This scheme is used to avoid the need to reload address mappings when context switching between tasks. The ASID in this register is checked against the ASID portion of a Page Descriptor (PD) unless the global bit, T, is set. Since there may be more than 256 tasks running at any one time, the operating system manages the allocation and use of ASIDs.

**Note**

The ASID is checked in both user and kernel mode — allowing the OS to run tasks in either mode.

Writes to the `PID` register should be made either from code running in un-translated memory or from code running from a page with the Global bit set (ASID is ignored). This approach ensures that the code page being accessed continues to be visible after the ASID is changed.

The processor ensures that the ASID update takes effect immediately after the SR instruction making the change.

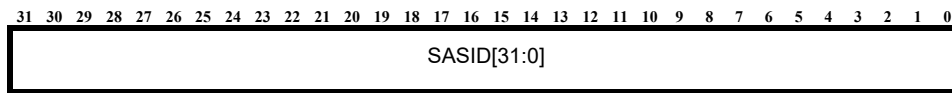
13.2.8 Shared Library Address Space Identifier Register, SASID0, SASID1

Address: 0x46A — SASID0
 0x46B — SASID1

Access: RW

Reset 0x00

Figure 13-8 Shared Library Address Space Identifier Register, SASID0, SASID1



This register is available in the processor only when `-mmu_shared_lib==true`.

The MMU supports shared libraries with a subscription scheme using the SASID register, allowing up to 64 shared libraries. SASID1 and SASID0 together form the 64-bit SASID register. Each bit position in the SASID register is associated with a shared library (a collection of pages) that a process is subscribed to. A 1 in a bit position indicates that the current process is subscribed to the associated library. The MMU page table entries with the S bit set use the entry's ASID field to determine if a matching VA corresponds to a shared library that the current process is subscribed to.

Each bit position in the SASID register is used to control the shared library that a process can subscribe to or use. A 1 in a bit position indicates that a process with a matching ASID (see [TLBPD0](#) and [PID](#)) can use the shared library.

The shared address space identifier register, SASID, is provided by MMU version 3 or higher; see [MMU_BUILD](#). The SASID register is configured when the SL bit is set in the [MMU_BUILD](#) register.

Shared Library ASID

For checking each way for the selected set, the global flag and PID/ASID match conditions are extended with the shared library ASID match. The global flag has the highest priority, followed by the shared library ASID match as the next priority, and finally the PID/ASID match that has the lowest priority.

If the shared flag is set for a page translation, the shared library ASID match takes priority over the PID/ASID match, and the PID/ASID match is ignored for the way.

A way hit of any type selects that way and declares the operation a hit. No hits in any way results in a miss. Multiple way hits in a set result in a Machine Check, Overlapping TLB Entries exception regardless of the type of match in each way.

Up to 64 concurrent shared library mappings are supported, with each library being allocated to a single bit of the SASID register. For a process to be subscribed to a particular shared library, set the bit in the SASID value for the process to 1. The exception is only raised in the case of instruction execution lookup (for fetch, load or store operations); for TLB maintenance operation lookups (probe, insert or delete) the `TLBIndex.E` and `RC` fields indicate overlapping entries.

When values are being looked up in the MMU, if the Shared (S) bit is set in an entry with a matching VPN, bits [5:0] of the ASID field of the matching entry are used as an index into the SASID register pair to evaluate whether the current context is subscribed to the shared library associated with the indexed bit. If the indexed bit is set, the entry is declared a match.

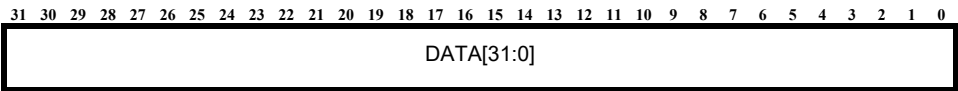
13.2.9 Scratch Data Register, SCRATCH_DATA0

Address: 0x46C

Access: RW

Reset 0x00

Figure 13-9 SCRATCH_DATA0 Register



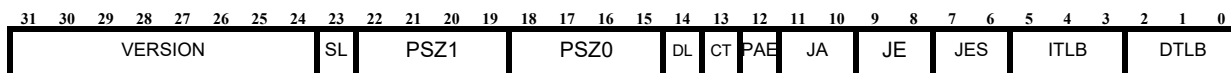
The SCRATCH_DATA0 auxiliary register is a generic 32-bit scratch register that can be used in kernel mode only to store any data. The OS can for example use this register to hold the base address of the first level page descriptor table to speed up page table access. The default on reset is 0x0 and writes to the SCRATCH_DATA0 are non-serializing.

13.3 MMU Build Configuration Register, MMU_BUILD

Address: 0x6F

Access: R

Figure 13-10 MMU_BUILD Register



The build configuration register `MMU_BUILD` (0x6F) contains information for the operating system to determine the configuration of the memory management unit. The TLB replacement algorithms are inferred from the version number.

Always access this register in kernel mode. An attempt to access this register from user mode raises a Privilege Violation exception.

Any write to the build configuration register, whether in kernel or user mode, raises an Illegal Instruction exception.

Table 13-5 MMU_BUILD Register Fields

Field		Description
DTLB	[2:0]	Indicates the number of DTLB/ μ DTLB entries 0x2: 8 entries 0x3: 16 entries All other values are reserved.
ITLB	[5:3]	Indicates the number of ITLB/ μ ITLB entries 0x1: 4 entries 0x3: 16 entries All other values are reserved.
JES	[7:6]	Joint super page TLB (STLB) contains S entries 0x0: 0 Entries 0x1: 16 Entries 0x2 to 0x3: Reserved The default is 0x0 (0 entries)

Table 13-5 MMU_BUILD Register Fields

Field		Description
JE	[9:8]	Joint Normal page TLB (NTLB) contains 2**(6+JE) entries per way. defaults to 0x0 (64 entries per way or 256 total entries) 0x0: 256 (4-way) entries or 32 (fully associative) 0x1: 512 entries (4-way) or 64 (fully associative) 0x2: 1024 entries (4-way) 0x3: reserved
JA	[11:10]	Joint NTLB contains 2**JA ways 0x0: fully associative 0x2: 4-way set associative All other values are reserved.
PAE	[12]	Physical Address Extension (40 bit) Enable 0x0: Disabled (32b physical address) 0x1: Enabled (40b physical address)
CT	[13]	0x0: CCM translation is not supported 0x1: CCM translation is supported
DL	[14]	0x0: Dynamic loading is not supported 0x1: Dynamic loading is supported.
PSZ0, PSZ1	[18:15] [22:19]	Page Size (PSZ0: normal page size and PSZ1: super page size) 0x0 to 0x2: Reserved 0xA: 512K bytes 0x3: 4K bytes 0xB: 1024K bytes 0x4: 8K bytes 0xC: 2M bytes 0x5: 16K bytes 0xD: 4M bytes 0x6: 32K bytes 0xE: 8M bytes 0x7: 64K bytes 0xF: 16M bytes 0x8: 128K bytes 0x9: 256K bytes
SL	[23]	Shared Library ASID When the Version field in this register is 0x3 or greater, this bit indicates that the shared-library address-space identifier registers, SASID0 , SASID1 are present and the shared-library ASID feature is available.
VERSION	[31:24]	VERSION 0x4: MMU Version 4 configuration with programmable page size selection, Physical Address Extension, new page attribute bit definitions. 0x5: MMU version 5 with support for dynamic loading.

14

ICCM

14.1 ICCM Auxiliary Registers

This section describes the ARCV2 Instruction Closely Coupled Memories (ICCM0 and ICCM1). ICCMs are designed to hold performance-critical code.

The ARCV2 architecture includes the following auxiliary registers and build configuration registers when the instruction close coupled memory is enabled. In builds which do not have any means to fetch instructions other than through the multiple ICCMs, the starting address of the ICCM0 is fixed at address 0.

Table 14-1 ICCM Registers

Address	Auxiliary Register Name	Description
0x208	AUX_ICCM	Base address of multiple ICCMs
0x78	ICCM_BUILD	Build configuration register for ICCM

14.1.1 ICCM base address, AUX_ICCM

Address: 0x208

Access: RW

Figure 14-1 AUX_ICCM, base address for ICCM (ADDR_SIZE == 32)



Figure 14-2 AUX_ICCM, base address for ICCM (ADDR_SIZE == 16)

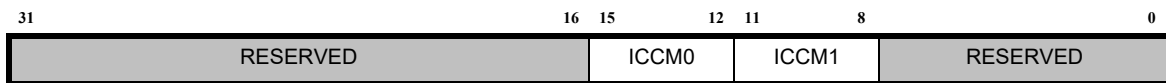
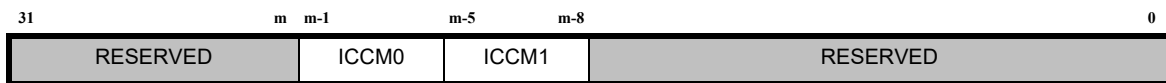


Figure 14-3 AUX_ICCM, base address for ICCM (ADDR_SIZE == m)



If an ICCM0 or ICCM1 is not configured in a build, the corresponding base address field of the AUX_ICCM register is reserved (RAZ/IOW). 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. 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]. 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 gets access to the regions where they overlap.

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.

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.



Caution

- Avoid mapping DCCM in the same region of the ICCMs. If the ICCM (ICCM0 or ICCM1) and DCCM are mapped to the same region, the transaction is handled by the ICCM.
- 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

14.1.2 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 14-4 ICCM_BUILD Register

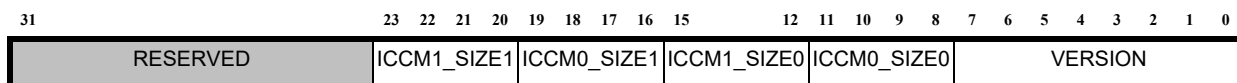


Table 14-2 ICCM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Current version 0x4
ICCM0_SIZE0	[11:8]	Size of ICCM0 RAM ■ 0x0: Not present ■ 0x1: 512B ■ 0x2: 1KB ■ 0x3: 2KB ■ 0x4: 4KB ■ 0x5: 8KB ■ 0x6: 16KB ■ 0x7: 32KB ■ 0x8: 64KB ■ 0x9: 128KB ■ 0xA: 256KB ■ 0xB: 512KB ■ 0xC: 1MB ■ 0xD: 2 MB ■ 0xE: 4 MB ■ 0xF: 8 MB

Table 14-2 ICCM_BUILD Field Descriptions (Continued)

Field	Bit	Description
ICCM1_SIZE0	[15:12]	Size of ICCM1 RAM ■ 0x0: Not present ■ 0x1: 512B ■ 0x2: 1KB ■ 0x3: 2KB ■ 0x4: 4KB ■ 0x5: 8KB ■ 0x6: 16KB ■ 0x7: 32KB ■ 0x8: 64KB ■ 0x9: 128KB ■ 0xA: 256KB ■ 0xB: 512KB ■ 0xC: 1MB ■ 0xD: 2 MB ■ 0xE: 4 MB ■ 0xF: 8 MB
ICCM0_SIZE1	[19:16]	When the ICCM0_Size0 == 0xF, ICCM0 SIZE = (8 MB * (2 ^ ICCM0_SIZE1)) ■ 0x0: 8 MB ■ 0x1: 16 MB ■ 0x2 to 0xF: reserved
ICCM1_SIZE1	[23:20]	When the ICCM1_Size0 == 0xF, ICCM1 SIZE = (8 MB * (2 ^ ICCM1_SIZE1)) ■ 0x0: 8 MB ■ 0x1: 16 MB ■ 0x2 to 0xF: reserved

14.2 Exceptions

See [Table 5-4](#) on page 233.

15

Instruction Cache

15.1 Instruction Cache Auxiliary Registers

The ARCV2 architecture includes additional auxiliary registers when the instruction cache is enabled. The number of additional auxiliary registers depend on the chosen `-ic_feature_level` configuration parameter. The IC_IVIC register is also included when an instruction fetch interface to external memory or ICCM1 is present in a processor. The instruction fetch path in some ARCV2-based processors may pre-fetch and cache a small number of instructions, which must be flushed when code memory is modified to ensure that stale instructions from the fetch queues are not executed. The IC_IVIC register is therefore included when either an instruction cache (I-cache) or a queuing interface to instruction memory is in the processor configuration. Writing to IC_IVIC invalidates the instruction fetch queues, instruction cache, or both.

Advanced Instruction-Cache Operation

The instruction cache contains advanced debug facilities that allow the programmer to interrogate and control the instruction cache RAM contents.

Advanced Features Summary

The instruction cache allows the programmer to obtain direct access to the internal cache RAMs through either the ARCV2 host interface bus or through LR and SR instructions. This capability provides an invaluable debug and test environment when using the instruction cache.

15.2 Auxiliary Registers

Table 15-1 Instruction Cache Auxiliary Registers

Address	Auxiliary Register Name	Description
0x10	Invalidate Instruction Cache, IC_IVIC	Invalidate instruction cache
0x11	Instruction Cache Control Register, IC_CTRL	Instruction cache control register
0x13	Lock Instruction Cache Line, IC_LIL	Lock instruction cache line
0x14	Instruction Cache Way-Lock Register, IC_WAYLOCK	Way-Lock Register

Table 15-1 Instruction Cache Auxiliary Registers (Continued)

Address	Auxiliary Register Name	Description
0x16	Invalidate Instruction-Cache Start Region, IC_IVIR	Instruction cache start region
0x17	Invalidate Instruction-Cache End Region, IC_ENDR	Instruction cache end region
0x19	Invalidate Instruction-Cache Line, IC_IVIL	Invalidate instruction cache line
0x1A	Instruction Cache External-Access Address, IC_RAM_ADDR	Instruction cache external access RAM address
0x1B	Instruction-Cache Tag Access, IC_TAG	Instruction cache tag access
0x1D	Instruction Cache Data Access, IC_DATA	Instruction cache data access
0x1E	Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG	Instruction Cache External Access RAM Address Physical Tag Format
0x1F	Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG_HI	Instruction Cache External Access RAM Address Physical Tag Format

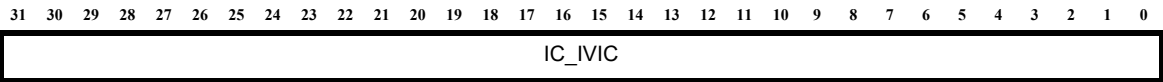
15.2.1 Invalidate Instruction Cache, IC_IVIC

Address: 0x10

Access: W

Reset: 0x0000_0000

Figure 15-1 IC_IVIC Register



A write to the IC_IVIC register invalidates the entire instruction cache. Builds with an instruction-fetch interface to external memory 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**

The IC_CTRL.SB bit is not updated when the instruction cache is successfully invalidated.

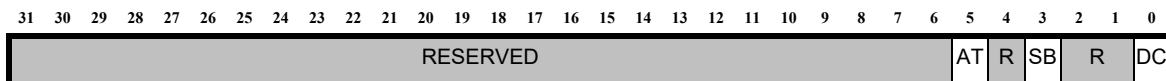
15.2.2 Instruction Cache Control Register, IC_CTRL

Address: 0x11

Access: RW

Reset: 0x0000_0000

Figure 15-2 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 15-2 IC_CTRL Control Flags

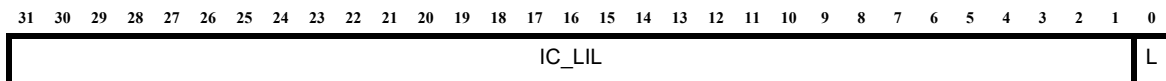
Flag Name	Bit	Reset	Description	Access Type
DC	[0]	0	Disable Cache: Enables/Disables the cache 0x0: Enable Cache 0x1: Disable Cache	R/W
SB	[3]	0	Success Bit: Success of last cache operation SB is for compatibility with L1 instruction or data cache. SB is set (SB=1) only for the following aux operations: - Line lock: if the line cannot be locked because all ways in the sets that the address maps to are already locked, the SB flag is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when the line lock operation completes successfully. - invalidates line or region: if the system memory address is present in the cache, the cache line is invalidated and unlocked; if the line is dirty and IM=1, it is flushed to the memory subsystem. When the line is present, the SB flag is set (SB=1). If the line is not present, the SB=0.	R/W
AT	[5]	0	Address Debug Type: Used for debug purposes for when accessing cache RAMs. 0x0: Direct Cache RAM Access 0x1: Cache Controlled RAM Access	R/W
RESERVED	[31:6]	0	Reserved. Read-as-zero; ignored on write.	R

For more information about the IC_CTRL register implementation, see your *ARCV2-based processor databook*.

15.2.3 Lock Instruction Cache Line, IC_LIL

Address: 0x13
 Access: W
 Reset: 0x0000_0000

Figure 15-3 IC_LIL Register



In non-aliased configurations, the IC_LIL register provides the full physical address (including the tag and index used in the cache lookup) of the line to be locked or invalidated.

In aliased instruction cache configurations this register holds only the index bits of the line address and the IC_PTAG register is used to program the physical page number (PPN) of the physical page containing the line to be locked. The IC_PTAG register must be setup first before triggering the operation with an SR to the IC_LIL register specifying the virtual index of the line to be locked. The physical byte address of the cache line is programmed in IC_LIL. Only address bits starting at bit $\log_2(\text{IC\_BSIZE})$ are significant. IC_BSIZE is the configured instruction cache line size.

When IC_LIL[0]==0

Writing a system-memory address to this register locks the cache line that maps to the given address. Locking allows the cache to be used as a high-speed instruction ROM that has a granularity of the cache-line size.

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.

- 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 is cleared (SB=0), indicating a cache operation failure. For the behavior of the SB flag for this operation, see the IC_CTRL.SB bit description. Note that if the programmer attempts to lock a previously locked line, the line is re-loaded and locked as normal.
- The instruction cache lock instructions operate on untranslated physical addresses. For aliased configurations, an SR to the IC_LIL register (lock) uses the virtual index from the address written to IC_LIL. No other bits from this register is used. The virtual index is combined with the physical page number in IC_PTAG to locate the cache line to be locked.

When IC_LIL[0]==1

Cache line prefetch (line is not locked). An SR instruction to IC_LIL executes a cache line prefetch. The cache line is fetched and stored in the instruction cache as a valid cache line. It replaces a cache line that is evicted based on the instruction cache replacement policy.

The physical byte address of the cache line is programmed in IC_LIL. Only address bits starting at bit $\log_2(\text{IC\_BSIZE})$ are significant. IC_BSIZE is the configured instruction cache line size.

It is possible that the prefetch operation fails when all ways of the instruction cache set to which the cache line is mapped are locked. In this case, the cache line for which the prefetch operation fails is not prefetched into the instruction cache.

The success or failure of a prefetch operation is indicated by IC_CTRL[SB]. When the cache line is successfully prefetched, IC_CTRL.SB is set to 1. When the prefetch fails then IC_CTRL.SB is set to 0.



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.

15.2.4 Invalidate Instruction-Cache Start Region, IC_IVIR

Address: 0x16

Access: W

Reset: 0x0000_0000

Figure 15-4 IC_IVIR Register



For non-aliased configurations, the IC_IVIR register provides the full physical address (including the tag and index fields) of the region start and end addresses of the region to be invalidated, prefetched, or locked. The starting address of the region is programmed in the upper bits of IC_IVIR, starting at bit $\log_2(\text{IC\_BSIZE})$. IC_BSIZE is the configured instruction cache line size.

When IC_IVIR[1:0]==0

When the processor is subject to VIPT aliasing in the instruction cache, the IC_PTAG register is also used in region operations. The region invalidate operation combines the PPN in IC_PTAG with the set indexes in IC_IVIR and IC_ENDR. No other bits in IC_IVIR and IC_ENDR registers are used. There is a single IC_PTAG register that provides the physical page number for both start and end of region. Therefore region invalidate cannot span a page boundary.

When a system memory address is written to this register, the cache region starting with the cache line that maps to the given address (IC_IVIR), including locked lines, is invalidated. The operation invalidates all lines matching addresses from the given address to the address previously written to the IC_ENDR (end region) address. The address specified by IC_ENDR is excluded. You must set the IC_ENDR register before writing to the IC_IVIR register.

The SB flag in IC_CTRL can be checked to verify the completion of an IC_IVIR operation. For the behavior of the SB flag for this operation, see the IC_CTRL.SB bit description.

When IC_IVIR[1:0]==1

Region prefetch. An SR instruction to IC_IVIR starts a region prefetch. All cache lines with physical addresses in the region are prefetched into the instruction cache.

It is possible that the prefetch operation fails for one or more instruction cache lines in the region when all ways of the instruction cache to which the cache line is mapped are locked. The cache line for which the prefetch operation fails is not prefetched into the instruction cache.

The success or failure of a region prefetch operation is indicated by $\text{IC\_CTRL}[\text{SB}]$. When all cache lines in the region are successfully prefetched, IC_CTRL.SB is set to 1. When the prefetch fails for any of the instruction cache lines in the region, IC_CTRL.SB is set to 0.

When IC_IVIR[1:0]==2

Region lock. An SR instruction to IC_IVIR starts a region lock. All cache lines with physical addresses in the region are prefetched into the instruction cache, and locked.

It is possible that the lock operation fails for one or more cache lines in the region. when all ways, of the instruction cache to which the cache line is mapped, are already locked. The cache line for which the lock operation fails is not prefetched into the instruction cache and not locked.

The success or failure of a region lock operation is indicated by the `IC_CTRL[SB]`. When all cache lines in the region are successfully prefetched and locked, `IC_CTRL.SB` is set to 1. When the lock fails for any of the icache lines in the region then `IC_CTRL.SB` is set to 0.

IC_IVIR[1:0]==3 is reserved

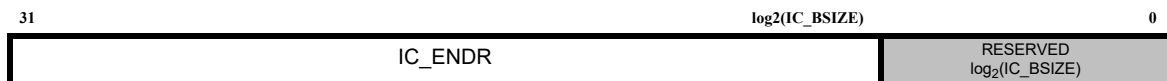
15.2.5 Invalidate Instruction-Cache End Region, IC_ENDR

Address: 0x17

Access: RW

Reset: 0x0000_0000

Figure 15-5 IC_ENDR Register



For non-aliased configurations, the IC_IVIR register provides the full physical address (including the tag and index fields) of the region start and end addresses of the region to be invalidated.

For aliased configurations (cache way size is larger than the MMU page size), this register holds only the index bits of the region end address and the IC_PTAG register is used to program the physical page number (PPN) of the physical page containing the region to be invalidated.

This register is used to hold the last program memory address up to which the instruction cache is invalidated. During an invalidate operation, the cache region starting from IC_IVIR to IC_ENDR is invalidated. The address specified by IC_ENDR is excluded. You must set this register before writing to the IC_IVIR register.

Note that the programming model is dependent on the size of the cache: a 64 kByte cache (with aliasing) is programmed differently than a 32 kByte cache (without aliasing).

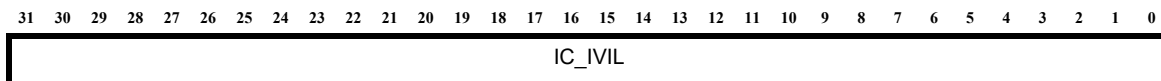
15.2.6 Invalidate Instruction-Cache Line, IC_IVIL

Address: 0x19

Access: W

Reset: 0x0000_0000

Figure 15-6 IC_IVIL Register



Typical uses of the cache invalidation are to update an interrupt vector without invalidating the whole cache, or for cache-locking maintenance. The SB flag in IC_CTRL can be checked to verify the completion of an IC_IVIL operation.

In non-aliased configurations, the IC_IVIL register provides the full physical address (including the tag and index used in the cache lookup) of the line to be locked or invalidated.

In aliased instruction cache configurations (cache way size is larger than the MMU page size), this register holds only the index bits of the line address and the IC_PTAG register is used to program the physical page number (PPN) of the physical page containing the line to be invalidated. The IC_PTAG register must be setup first before triggering the operation with an SR to the IC_IVIL register specifying the virtual index of the line to be invalidated. The instruction cache invalidate instructions operate on untranslated physical addresses. For aliased configurations, an SR to the IC_IVIL register (invalidate) uses the virtual index from the address written to IC_IVIL. No other bits from this register is used. The virtual index is combined with the physical page number in IC_PTAG to locate the cache line to be invalidated.

When a system memory address is written to this register the cache invalidates a cache line that maps to the given address, including locked lines. If the location specified is not in the cache, the SB flag in IC_CTRL is cleared (SB=0). Otherwise, the SB bit is set (SB=1).

Table 15-5 IC_RAM_ADDRESS Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the MMU page size)
Direct Mode (AT==0)	■ IC_PTAG is not used ■ SR to IC_RAM_ADDR: - triggers a read from RAM - WAY/INDEX/OFFSET specified by IC_RAM_ADDR - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM, but the index field of IC_TAG is used. IC_DATA is updated with the data value from the instruction cache data RAM ■ SR to IC_TAG updates the IC_TAG register and has the side-effect of writing the new value to the tag RAM at the location specified by the way and index field in IC_RAM_ADDR. ■ SR to IC_DATA updates the IC_DATA register and has the side-effect of writing to the new value to the instruction cache data RAM at the location specified by way, index and offset in IC_RAM_ADDR. ■ LR from IC_RAM_ADDR has no side-effect and only returns the value in that register. ■ LR from IC_TAG and IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs.	Same as non-aliased except the index field of IC_TAG is not used (remains 0).

Table 15-5 IC_RAM_ADDRESS Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the MMU page size)
Cache Controlled Mode (AT==1)	■ SR to IC_RAM_ADDR: - triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address; in this case, the physical address is specified in the IC_RAM_ADDR - Selects set and way matching the given address - updates IC_CTRL.SB with cache hit status - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM. IC_DATA is updated with the data value from the instruction cache data RAM; the index field of IC_TAG is updated with a copy from IC_RAM_ADDR. ■ SR to IC_TAG or IC_DATA triggers an instruction error exception. ■ LR from IC_TAG and IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs.	■ SR to IC_RAM_ADDR: - The IC_PTAG register must be programmed first with the physical page number. - Then the IC_RAM_ADDR must be programmed with the virtual index and offset; this triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address but tag matching uses the full PPN - updates IC_CTRL.SB with the cache hit status - If cache hit: IC_TAG gets the physical tag, lock and valid bits, but not the index. IC_DATA gets the data word from the instruction cache data RAM. IC_PTAG is unchanged ■ SR to IC_TAG or IC_DATA triggers an instruction error exception ■ LR from IC_TAG and IC_DATA only retrieves the values in those registers, but doesn't trigger an update of these registers with values from the tag and data RAMs

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.

15.2.8 Instruction-Cache Tag Access, IC_TAG

Address: 0x1B

Access: RW

Reset: 0x0000_0000

Figure 15-8 IC_TAG Register for Reads

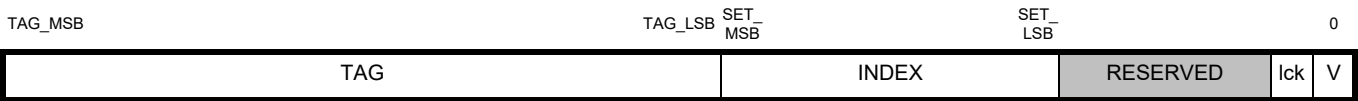


Table 15-6 IC_TAG Register for Reads in an Aliased Configuration

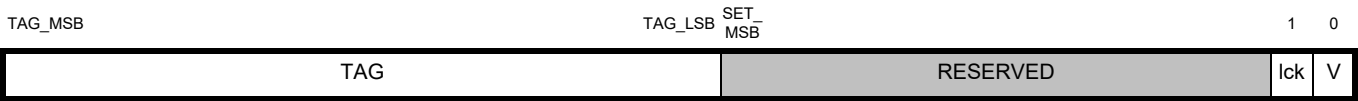


Figure 15-9 IC_TAG Register for Writes



 **Note** Following are the definitions of the various fields in this register:

- SET_LSB = $\text{Log}_2(\text{IC\_BSIZE})$
- SET_MSB = $\text{Log}_2(\text{IC\_SIZE}/\text{IC\_WAYS}) - 1$
- TAG_LSB = $\text{Log}_2(\text{IC\_SIZE}/\text{IC\_WAYS})$
- TAG_MSB = $\text{PC\_SIZE} - 1$

Reading from or writing to this register allows the programmer to access the valid bit, lock bit, and tag of a cache line specified by the IC_RAM_ADDR register and the AT flag found in IC_CTRL. An SR to the IC_RAM_ADDRESS register also triggers an update of the IC_TAG and IC_DATA registers.

When the physical address extension is enabled in the MMU, the IC_PTAG_HI register is also updated with the upper bits[39:32] of the tag.

Table 15-7 describes the IC_TAG register behavior.

Table 15-7 IC_TAG Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the MMU page size)
Direct Mode (AT==0)	■ SR to IC_TAG updates the IC_TAG register and has the side-effect of writing the new value to the tag RAM at the location specified by the way and index field in IC_RAM_ADDR ■ LR from IC_TAG only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ IC_PTAG is not used. ■ SR to IC_RAM_ADDR: - triggers a read from RAM - WAY/INDEX/OFFSET specified by IC_RAM_ADDR - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM, but the index field of IC_TAG is used. IC_DATA is updated with the data value from the instruction cache data RAM	Same as non-aliased except the index field of IC_TAG is not used (remains 0)

Table 15-7 IC_TAG Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the MMU page size)
Cache Controlled Mode (AT==1)	■ SR to IC_TAG triggers an instruction error exception ■ LR from IC_TAG only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ SR to IC_RAM_ADDR: - triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address; in this case, the physical address is specified in the IC_RAM_ADDR - Selects set and way matching the given address - updates IC_CTRL.SB with cache hit status - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM. IC_DATA is updated with the data value from the instruction cache data RAM; the index field of IC_TAG is updated with a copy from IC_RAM_ADDR.	■ SR to IC_TAG triggers an instruction error exception ■ LR from IC_TAG only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ SR to IC_RAM_ADDR: - The IC_PTAG register must be programmed first with the physical page number. - Then the IC_RAM_ADDR must be programmed with the virtual index and offset; this triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address but tag matching uses the full PPN - updates IC_CTRL.SB with the cache hit status - If cache hit: IC_TAG gets the physical tag, lock and valid bits, but not the index. IC_DATA gets the data word from the instruction cache data RAM. IC_PTAG is unchanged

**Caution**

Take care when debugging any software that accesses the TAG RAM via the IC_TAG register.

As part of the host debugger protocol to access memory, an invalidate-cache operation is carried out after any write to memory. This invalidate cache operation updates tag RAMs and therefore modifies any value that has been written to the tag RAM by the software.

An invalidate cache operation goes ahead even if the instruction cache is disabled.

Valid Bit (V[0])

The valid bit (V) indicates whether the data associated with the tag is valid or not. When V=1 the data is valid.

Lock Bit (LCK[1])

The lock bit (LCK) indicates whether the data associated with the tag is locked. The data entry is locked when LCK=1.

When the Physical Address Extension is enabled in the MMU, the IC_PTAG_HI register is also updated with the upper bits[39:32] of the tag.

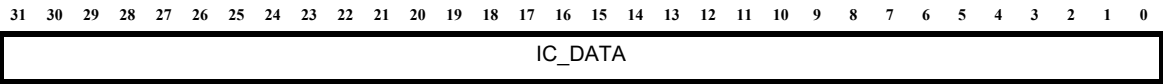
15.2.9 Instruction Cache Data Access, IC_DATA

Address: 0x1D

Access: RW

Reset: 0x0000_0000

Figure 15-10 IC_DATA Register



Reading from or writing to the IC_DATA register allows the programmer to access a data word from a particular cache line, using the address in IC_RAM_ADDR. An SR to the IC_RAM_ADDRESS also triggers an update of the IC_TAG and IC_DATA registers.

Table 15-8 describes the behavior of the IC_DATA register in a RAM direct access mode and a cache controlled mode.

Table 15-8 IC_DATA Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the MMU page size)
Direct Mode (AT==0)	■ SR to IC_DATA updates the IC_DATA register and has the side-effect of writing to the new value to the instruction cache data RAM at the location specified by way, index, and offset in IC_RAM_ADDR ■ LR from IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ IC_PTAG is not used ■ SR to IC_RAM_ADDR: - triggers a read from RAM - WAY/INDEX/OFFSET specified by IC_RAM_ADDR - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM, but the index field of IC_TAG is used. IC_DATA is updated with the data value from the instruction cache data RAM	Same as non-aliased except the index field of IC_TAG is not used (remains 0)

Table 15-8 IC_DATA Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the MMU page size)
Cache Controlled Mode (AT==1)	■ SR to IC_DATA triggers an instruction error exception ■ LR from IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ SR to IC_RAM_ADDR: - triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address; in this case, the physical address is specified in the IC_RAM_ADDR - Selects set and way matching the given address - updates IC_CTRL.SB with cache hit status - updates IC_TAG with the physical tag, lock bit and valid bit from the tag RAM. IC_DATA is updated with the data value from the instruction cache data RAM; the index field of IC_TAG is updated with a copy from IC_RAM_ADDR	■ SR to IC_DATA triggers an instruction error exception ■ LR from IC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs ■ SR to IC_RAM_ADDR: - The IC_PTAG register must be programmed first with the physical page number. - Then the IC_RAM_ADDR must be programmed with the virtual index and offset; this triggers a read from RAMs similar to a normal fetch access searching for an instruction word at a given address but tag matching uses the full PPN - updates IC_CTRL.SB with the cache hit status - If cache hit: IC_TAG gets the physical tag, lock and valid bits, but not the index. IC_DATA gets the data word from the instruction cache data RAM. IC_PTAG is unchanged

15.2.10 Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG

Address: 0x1E
 Access: RW
 Reset: 0x0000_0000

Figure 15-11 IC_PTAG Register



When the ARCV2 processor has an instruction cache and OLM, and the instruction cache way size (depth) is larger than the smallest page size (size0), there is the possibility of cache aliasing due to the VIPT Implementation of instruction cache. In this case, the IC_PTAG register is present in the build and is used to specify the tag field of addresses for cache controlled mode operations and cache line operations (lock, invalidate). In the aliased configuration (cache way size is larger than the OLM page size), IC_PTAG [and IC_PTAG_HI, if the physical address extension is enabled] is used for the physical page number, before the virtual address (index) is written into the respective IC_* aux register.

A write to IC_PTAG sets the value of the IC_PTAG register, but has no other side-effect and does not trigger a write to the TAG RAMs or other parts of the instruction cache.

A read from IC_PTAG returns the value of IC_PTAG, but has no other side-effect and does not trigger a read from the TAG RAMs or other parts of the instruction cache.

When the ARCV2 processor has an instruction cache and MMU, and the instruction cache way size (depth) is larger than the smallest page size (size0), there is the possibility of cache aliasing due to the VIPT Implementation of instruction cache. In this case, the IC_PTAG register is present in the build and is used to specify the tag field of addresses for cache controlled mode operations and cache line operations (lock, invalidate). In the aliased configuration (cache way size is larger than the MMU page size), IC_PTAG [and IC_PTAG_HI, if the physical address extension is enabled] is used for the physical page number, before the virtual address (index) is written into the respective IC_* aux register.

A write to IC_PTAG sets the value of the IC_PTAG register, but has no other side-effect and does not trigger a write to the TAG RAMs or other parts of the instruction cache.

A read from IC_PTAG returns the value of IC_PTAG, but has no other side-effect and does not trigger a read from the TAG RAMs or other parts of the instruction cache.

When the physical address extension is enabled, regardless of cache or MMU page sizes, the IC_PTAG_HI register exists and provides physical address bits above bit 31. This is true for line operations as well as region operations and serves tag matching as well as providing the upper address bits for external bus access; for region operations, since both region start and region end addresses use this register for upper address bits (39:32), a region cannot span a 4G address boundary.

15.2.11 Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG_HI

Address: 0x1F

Access: RW

Reset: 0x0000_0000

Figure 15-12 IC_PTAG_HI Register



When the ARCV2 processor has an instruction cache and MMU, and the instruction cache way size (depth) is larger than the smallest page size (size0), there is the possibility of cache aliasing due to the VIPT Implementation of instruction cache. In this case, the IC_PTAG register is present in the build and is used to specify the tag field of addresses for cache controlled mode operations and cache line operations (lock, invalidate). In the aliased configuration (cache way size is larger than the MMU page size), IC_PTAG [and IC_PTAG_HI, if the physical address extension is enabled] is used for the physical page number, before the virtual address (index) is written into the respective IC_* aux register.

A write to IC_PTAG sets the value of the IC_PTAG register, but has no other side-effect and does not trigger a write to the TAG RAMs or other parts of the instruction cache.

A read from IC_PTAG returns the value of IC_PTAG, but has no other side-effect and does not trigger a read from the TAG RAMs or other parts of the instruction cache.

When the physical address extension is enabled, regardless of cache or MMU page sizes, the IC_PTAG_HI register exists and provides physical address bits above bit 31. This is true for line operations as well as region operations and serves tag matching as well as providing the upper address bits for external bus access; for region operations, since both region start and region end addresses use this register for upper address bits (39:32), a region cannot span a 4G address boundary.

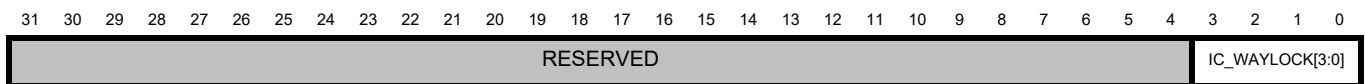
15.2.12 Instruction Cache Way-Lock Register, IC_WAYLOCK

Address: 0x14

Access: RW

Reset: 0x00000000

Figure 15-13 IC_WAYLOCK Register



When a 2-way set-associative instruction cache is configured, IC_WAYLOCK is fixed to 0.

When a 4-way set-associative instruction cache is configured

Use the IC_WAYLOCK register to determine which ways can be locked and not replaced. A 4-way set-associative instruction cache can be partitioned by selectively locking the ways of the sets. A waylock mechanism can be activated to restrict the ways of each set where cache lines can be replaced when an instruction cache miss occurs. Replacement in each set is restricted to unlocked ways. A waylock applies to the whole cache. For example, if ways 0 and 1 are locked by a waylock, cache replacement can only occur in ways 2 and 3. Instruction cache hits are serviced from any location in the instruction cache where the cache hit occurs and is not dependent on waylocks.

An SR write instruction to IC_WAYLOCK takes effect without a processor restart. Instruction cache misses that occur after the SR instruction commits take the updated value of the IC_WAYLOCK register into consideration. Note that instruction cache misses are handled by the processor early in the pipeline. There is pipeline delay after instruction fetch of the SR instruction during which subsequent instructions are fetched. This scenario might trigger instruction cache misses that still use the prior value of the IC_WAYLOCK register. That pipeline delay is variable.

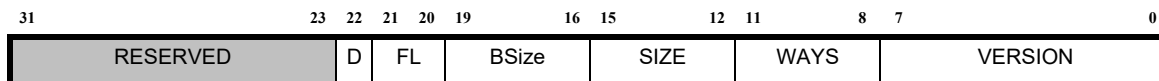
Each bit in the IC_WAYLOCK[3:0] field represents a way. To lock way 0, set the IC_WAYLOCK[0]=1. Similarly, to lock a way, write 1 to the corresponding bit. Each way can be locked independently of the other ways.

15.2.13 Instruction Cache Configuration Register, I_CACHE_BUILD

Address: 0x77

Access: R

Figure 15-14 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 15-9 I_CACHE_BUILD Field Descriptions

Field Name	Bit	Description
VERSION	[7:0]	Version number 0x0: No I_CACHE_BUILD register 0x1: Reserved 0x2: ARCompact V1, fixed 32-byte line size 0x3: ARCompact V1, variable line size 0x4: ARCV2 0x6: Includes the following enhancements: - Support for instruction cache line prefetch in the IC_LIL register - Support for the region prefetch and lock in the IC_IVIR register. - Support for the instruction cache waylock register <i>All other values are Reserved</i>
WAYS	[11:8]	Cache Associativity 0x1: Two-way set associative 0x2: Four-way set associative <i>All other values are Reserved</i>
SIZE	[15:12]	Cache capacity 0x2: 2 Kbytes 0x3: 4 Kbytes 0x4: 8 Kbytes 0x5: 16 Kbytes 0x6: 32 Kbytes 0x7: 64 Kbytes <i>All other values are Reserved</i>

Table 15-9 I_CACHE_BUILD Field Descriptions (Continued)

Field Name	Bit	Description
BSize	[19:16]	Block Size, indicates the cache block size in bytes. ■ 0x2: 32 bytes ■ 0x3: 64 bytes ■ 0x4: 128 bytes <i>All other values are Reserved</i>
FL	[21:20]	Feature Level ■ 0x2: Line lock, invalidate, and advanced debug features are supported ■ 0x3: Reserved
D	[22]	Indicates if IC is disabled on reset. ■ 0x0: indicates that IC is not disabled on reset ■ 0x1: indicates that IC is disabled on reset
RESERVED	[31:23]	Reserved. Read-as-zero; ignored on write.

15.3 Exceptions

See [Table 5-4](#) on page 233.

16

DCCM

16.1 DCCM Auxiliary Registers

This section describes the ARCV2 Data Closely Coupled Memories (DCCM). DCCM is designed to hold performance-critical data.

The ARCV2 architecture includes the following auxiliary registers and build configuration registers when the DCCM is enabled.

Table 16-1 DCCM Auxiliary Registers

Address	Auxiliary Register Name	Description
0x18	DCCM Base Address, AUX_DCCM	Start address of the DCCM
0x74	DCCM Configuration Register, DCCM_BUILD	DCCM build configuration register

16.1.1 DCCM Base Address, AUX_DCCM

Address: 0x18

Access: RW

Default 0x8

Figure 16-1 AUX_DCCM, base address for DCCM (ADDR_SIZE == 32)



Figure 16-2 AUX_DCCM, base address for DCCM (ADDR_SIZE == 16)

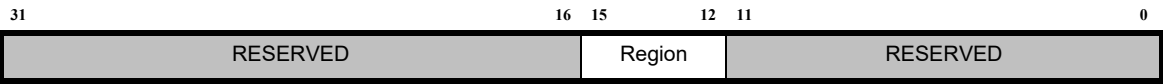


Figure 16-3 AUX_DCCM, base address for DCCM (ADDR_SIZE == m)



When a DCCM is configured in a processor, the AUX_DCCM register identifies the region in which the DCCM is contained.

If DCCM is not configured, this register does not exist. The most-significant relevant bit of AUX_DCCM is determined by the configured size of the address bus, ADDR_SIZE, such that the 4-bit Region number for the DCCM is always located at positions [ADDR_SIZE-1:ADDR_SIZE-4].

**Caution**

- Avoid mapping DCCM in the same region of the ICCMs. If the ICCM (ICCM0 or ICCM1) and DCCM are mapped to the same region, the transaction is handled by the ICCM.
- 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
- DCCM is not visible to the Instruction Fetch Unit (IFU), and so an instruction fetch to an address within the DCCM region may target the external memory (assuming ICCMs are not mapped there). In an MMU configuration, DCCM is invisible to IFU independent of whether the address (VA) is translated (lower 2G) or untranslated (upper 2G).
- If DCCM is placed in the translated space (in an MMU configuration), data (ld/st) accesses to DCCM are checked for permissions but addresses within the DCCM are not be altered by the translation (and detection of the DCCM 'hole' is unaffected by translation).

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.

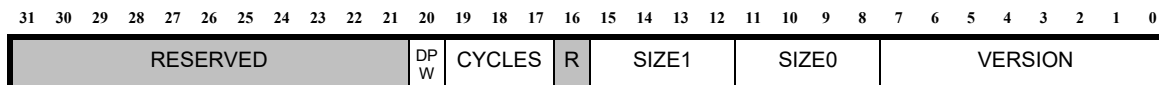
You can configure the DCCM region using the ARChitect tool; on reset, the `Region` field is initialized to this value.

16.1.2 DCCM Configuration Register, DCCM_BUILD

Address: 0x74

Access: R

Figure 16-4 DCCM_BUILD Register



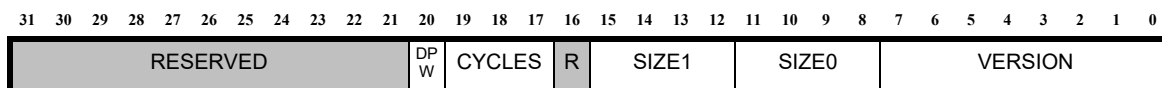
The DCCM RAM Configuration register (DCCM_BUILD) describes the size and version number of the Data Closely Coupled Memory.

Table 16-2 DCCM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Current version: 0x4; DCCM with configurable number of cycles. 0x3: DCCM with fixed number of cycles. Other values are reserved.
SIZE0	[11:8]	Size of DCCM RAM ■ 0x0: Not present ■ 0x1: 512B ■ 0x2: 1KB ■ 0x3: 2KB ■ 0x4: 4KB ■ 0x5: 8KB ■ 0x6: 16KB ■ 0x7: 32KB ■ 0x8: 64KB ■ 0x9: 128KB ■ 0xA: 256KB ■ 0xB: 512KB ■ 0xC: 1MB ■ 0xD: 2MB ■ 0xE: 4 MB ■ 0xF: 8 MB

Table 16-2 DCCM_BUILD Field Descriptions (Continued)

Field	Bit	Description
SIZE1	[15:12]	When the Size0 == 0xF, DCCM Size = (8 MB * (2 ^ SIZE1)) 0x0: 8 MB 0x1: 16 MB 0x2 to 0xF: reserved
CYCLES	[19:17]	Two-cycle or single-cycle DCCM 0x0: Not present 0x1: 1 cycle – single cycle memory; the number of memory banks (2 or 4) can be configured using the <code>-dccm_mem_banks</code> configuration option. DCCM banks are 32-bit wide (at 4-byte aligned addresses) with interleaved addresses. 0x2: 2 cycles – two-cycle memory; always includes four banks that are 32-bit wide (at 4-byte aligned addresses) with interleaved addresses. All other values are reserved. Note: This field is valid only when <code>DCCM_BUILD.VERSION==0x4</code>.
DPW	[20]	Indicates whether partial writes through the DMI interface is configurations with ECC/parity protection. 0x0: Not configured. A partial DCCM DMI write triggers a bus error. 0x1: Configured This field reflects the value of the configuration option: <code>-dccm_dmi_partial_ecc</code>

Figure 16-5 DCCM_BUILD Register

The DCCM RAM Configuration register (DCCM_BUILD) describes the size and version number of the Data Closely Coupled Memory.

Table 16-3 DCCM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	0x3: DCCM with fixed number of cycles. 0x4: Current version: DCCM with configurable number of cycles. Other values are reserved.

Table 16-3 DCCM_BUILD Field Descriptions (Continued)

Field	Bit	Description
SIZE0	[11:8]	Size of DCCM RAM 0x0: Not present 0x1: 512B 0x2: 1KB 0x3: 2KB 0x4: 4KB 0x5: 8KB 0x6: 16KB 0x7: 32KB 0x8: 64KB 0x9: 128KB 0xA: 256KB 0xB: 512KB 0xC: 1MB 0xD: 2MB 0xE: 4 MB 0xF: 8 MB
SIZE1	[15:12]	When the Size0 == 0xF, DCCM Size = (8 MB * (2 ^ SIZE1)) 0x0: 8 MB 0x1: 16 MB 0x2 to 0xF: reserved
CYCLES	[19:17]	Two-cycle or single-cycle DCCM 0x0: Not present 0x1: 1 cycle – single cycle memory; the number of memory banks (2 or 4) can be configured using the <code>-dccm_mem_banks</code> configuration option. DCCM banks are 32-bit wide (at 4-byte aligned addresses) with interleaved addresses. 0x2: 2 cycles – two-cycle memory; always includes four banks that are 32-bit wide (at 4-byte aligned addresses) with interleaved addresses. All other values are reserved. Note: This field is valid only when <code>DCCM_BUILD.VERSION==0x4</code> .

Table 16-3 DCCM_BUILD Field Descriptions (Continued)

Field	Bit	Description
DPW	[20]	Indicates whether partial writes through the DMI interface is supported in configurations with ECC/parity protection. 0x0: Not supported. A partial DCCM DMI write triggers a bus error. 0x1: Supported This field reflects the value of the configuration option: -dccm_dmi_partial_ecc

16.1.3 Exceptions

See [Table 5-4](#) on page 233.

17

Data Cache

The ARCV2 architecture includes additional auxiliary registers when the data cache component is enabled. For ARC HS family of processors, `-dc_feature_level` is always set to 2, and is not configurable.

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.

17.1 Data Cache Auxiliary Registers

Table 17-1 Data Cache Auxiliary Registers

Address	Register Name ¹	Description	Initial Value
0x47	Invalidate Data Cache, DC_IVDC	Invalidate cache	-
0x48	Data Cache Control Register, DC_CTRL	Data cache control register	0xC0
0x4B	Flush Data Cache, DC_FLSH	Flush data cache	-
0x5E	AUX_VOLATILE	Defines the volatile uncached space in the memory space	Defined by the following configuration options: ■ <code>-aux_volatile_base</code> ■ <code>-aux_volatile_limit</code> ■ <code>aux_volatile_disable</code> ■ <code>-aux_volatile_strict_ordering</code>
0x49	Lock Data Cache Line, DC_LDL	Lock data line	-
0x4A	Invalidate Data Line, DC_IVDL	Invalidate data line	-
0x4C	Flush Data Line, DC_FLDL	Flush data line	-

Table 17-1 Data Cache Auxiliary Registers

Address	Register Name ¹	Description	Initial Value
0x4D	Data-Cache Region Start Address, DC_STARTR	Data cache region operation start address	-
0x4E	Data-Cache Region End Address, DC_ENDR	Data cache region operation end address	-
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
0x5F	Data Cache External Access RAM Address Physical Tag Format, DC_PTAG_HI	Data cache tag access	0x0000 0000
0x5B	Data Cache Data Access, DC_DATA	Data cache data access	Undefined

1. All writes to the data cache maintenance auxiliary registers are serializing and stall a following NOP for at least 11 cycles.

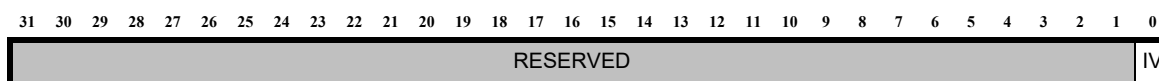
17.1.1 Invalidate Data Cache, DC_IVDC

Address: 0x47

Access: W

Reset: 0x0000_0000

Figure 17-1 DC_IVDC Register



DC_IVDC is typically used to reset the data-cache contents upon a software process switch.

Table 17-2 DC_IVDC Field Descriptions

Flag Name	Bit	Reset	Description	Access Type
IV	[0]	0	Invalidate Data Cache: Invalidates entire data cache 0x0: No Action 0x1: Invalidate Data Cache	W
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.	W

Writing a 1 to the IV flag (bit 0) in the DC_IVDC register invalidates and unlocks the entire data cache.

Upon a write to the IV flag, the operation of the invalidate function is determined by the Invalidate Mode (IM) flag that is found in DC_CTRL (bit 6).

All cache lines are invalidated and unlocked, and each line entry is processed according to the state of the IM flag.

The IM flag (bit 6, DC_CTRL) controls whether a cache line entry that is marked as being dirty is flushed, invalidated, and unlocked or simply invalidated and unlocked when a DC_IVDC command is issued.

- IM = 0 (Invalidate Only) — Any write to the DC_IVDC register invalidates all cache-line entries.
- IM = 1 (Invalidate and Flush Dirty Entries) — Any write to the DC_IVDC register flushes all dirty lines and invalidates the entire data cache. The programmer must be aware that in many cases, setting the IM flag to 1 and initiating a DC_IVDC command results in an indeterminate cache-cycle time stall. The cycle-time latency is dependent on the number of entries that are dirty and the efficiency of the memory subsystem.



After triggering the cache operation (full, line or region) software needs to wait for operation to complete by polling DC_CTRL.FS bit since the operation could potentially take many cycles, depending on the number of valid or dirty lines in the cache.

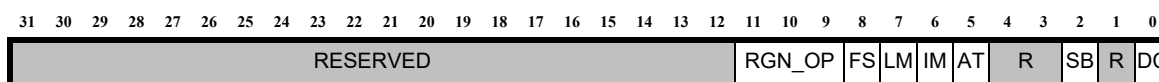
17.1.2 Data Cache Control Register, DC_CTRL

Address: 0x48

Access: RW

Reset: 0x0000_00c0

Figure 17-2 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 17-3 DC_CTRL Control Flags

Flag Name	Bit	Reset	Description	Access Type
DC	[0]	0	Disable Cache: Enables/Disables the cache 0 – Enable Cache 1 – Disable Cache	R/W
RESERVED	[1]	0	Reserved. Read-as-zero; ignored on write.	R/W
SB	[2]	0	Success Bit: Success of last cache operation SB is for compatibility with L1 instruction or data cache. SB is set (SB=1) only for the following aux operations: - Line lock: if the line cannot be locked because all ways in the sets that the address maps to are already locked, the SB flag is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when the line lock operation completes successfully. - invalidates line or region and flushes dirty entry: if the system memory address is present in the cache, the cache line is invalidated and unlocked; if the line is dirty and IM=1, it is flushed to the memory subsystem. When the line is present, the SB flag is set (SB=1). If the line is not present, the SB=0. - flush line or region: if the system memory address is present in the cache, the SB flag is set (SB=1) upon completion regardless of whether the dirty bit is present. Cache flush does not set the SB bit.	R/W
RESERVED	[4:3]	0	Reserved. Read-as-zero; ignored on write.	R/W
AT	[5]	0	Address Debug Type: Used for debug purposes for when accessing cache RAMs. ■ 0x0: Direct Cache RAM Access ■ 0x1: Cache Controlled RAM Access	R/W

Table 17-3 DC_CTRL Control Flags

Flag Name	Bit	Reset	Description	Access Type
IM	[6]	1	Invalidate Mode: Selects the invalidate type for line operations (DC_IVDL), entire data cache (DC_IVDC), and region operations (DC_CTRL[RGN_OP], DC_STARTR, DC_ENDR). 0x0: Invalidate data cache only 0x1: Flush dirty entries and invalidate data cache and this operation is also referred to as a purge operation.	R/W
LM	[7]	1	Lock Mode: Selects the effect of a flush command on a locked entry 0x0: Disable flush on locked entry 0x1: Enable flush on locked entry	R/W
FS	[8]	0	Flush Status: Status of the data-cache flush mechanism 0x0: Idle 0x1: Flush operation in progress	R
RGN_OP	[11:9]	0	Region Operation. Selects the operation to be performed on the address region specified by the DC_STARTR and DC_ENDR registers. This register and the DC_ENDR register must be set up first before writing DC_STARTR with the region start address which triggers the operation. 0x0: Region flush 0x1: Region invalidate; when this bit is programmed, the DC_CTRL[IM] bit is checked. If DC_CTRL[IM]==1, then the data cache region is flushed and then invalidated. 0x2 ~ 0x7: Reserved	R/W
RESERVED	[31:12]	0	Reserved. Read-as-zero; ignored on write.	R/W

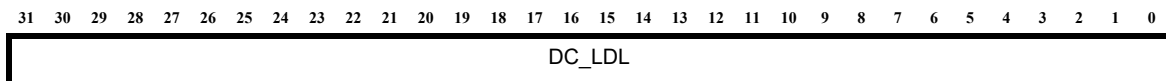
17.1.3 Lock Data Cache Line, DC_LDL

Address: 0x49

Access: W

Reset: 0x0000_0000

Figure 17-3 DC_LDL Register



Writing a system memory address to this register locks the cache line that maps to the given address. The format of the DC_LDL register is shown in [Figure 17-3](#).

For more information, see your ARCV2-based processor databook.

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.

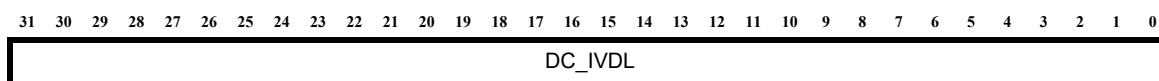
17.1.4 Invalidate Data Line, DC_IVDL

Address: 0x4A

Access: W

Reset: 0x0000_0000

Figure 17-4 DC_IVDL Register



Typical uses are to update a data entry within a table without invalidating the entire cache or use in cache-locking maintenance (such as removing cache-address mappings from the cache). Using this function also aids the update and maintenance of shared memory architectures where addresses within a memory space are shared between multiple logic blocks (such as other processors or peripherals).

When a system-memory address is written to this register, the cache invalidates the line that maps to the given address. If the line is dirty, it might be flushed before invalidation, depending on the IM bit. Locked lines are unlocked and invalidated just like unlocked lines.

The invalidate-data-line (DC_IVDL) operation allows a cache line to be removed from the cache. The method used to invalidate is selected via the IM flag (bit 6 of DC_CTRL).

- **IM = 0 (Invalidate Only)** – If the system memory address written to DC_IVDL is present in the cache, the cache line is invalidated and unlocked (if locked). For information about the SB flag for this operation, see the DC_CTRL.SB bit description.
- **IM = 1 (Invalidate and Flush Dirty Entry, also called as a purge operation)** – If the system memory address written to DC_IVDL is present in the cache, the cache line is invalidated and unlocked (if locked), and if the line is dirty it is flushed to the memory subsystem. For information about the SB flag for this operation, see the DC_CTRL.SB bit description.

17.1.5 Flush Data Cache, DC_FLSH

Address: 0x4B

Access: W

Reset: 0x0000_0000

Figure 17-5 DC_FLSH Register

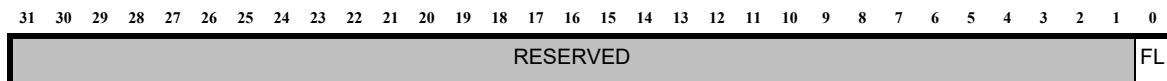


Table 17-4 DC_FLSH Control Flags

Flag Name	Bit	Reset	Description	Access Type
FL	[0]	0	Flush Data Cache:- Flush entire data cache 0x0: No Action 0x1: Flush Data Cache	W
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.	W

The DC_FLSH function is used to update main memory with the most recent copy of the data-cache contents. This is very useful for shared-memory architectures, where other processors require updated memory contents.

Writing a 1 to the FL flag in the DC_FLSH register initiates a flush sequence for the entire data cache. All cache sets are checked sequentially for dirty entries, and entries that contain modified data is flushed to the memory subsystem. Upon completion of a flush sequence the line-dirty bit is cleared.

The operation of a flush command on a locked entry is selected by the LM flag (bit 7) in DC_CTRL. Two sequences are possible with a locked entry.

- **LM = 0 (Disable Flush on Locked Entry)** — Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. A locked cache-line that contains a dirty entry is not flushed, keeping the cache line completely locked down.
- **LM = 1 (Enable Flush on Locked Entry)** — Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. A locked cache line that contains a dirty entry is also flushed irrespective of its locked status.

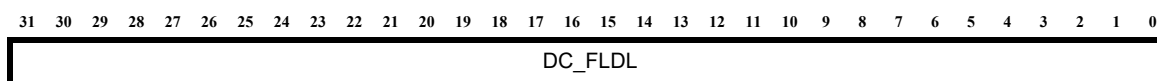
17.1.6 Flush Data Line, DC_FLDL

Address: 0x4C

Access: W

Reset: 0x0000_0000

Figure 17-6 DC_FLDL Register



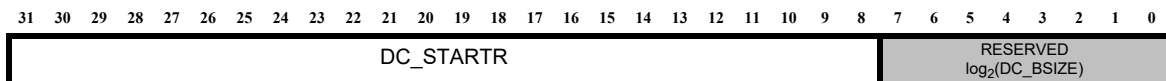
Writing a system-memory address to DC_FLDL causes the corresponding cache-line entry to flush (if dirty). If a cache-line entry is present and is not dirty, the cache-line entry is not flushed. In the event that an entry is locked, the state of the LM flag (bit 7, DC_CTRL) controls the effect of a DC_FLDL command on the locked entry.

- **LM = 0 (Disable Flush on Locked Entry)** — A system-memory address written to the DC_FLDL register initiates a search for that entry. In the event that the requested cache line is locked, the DC_FLDL command does not generate a flush sequence, even if that entry is dirty. For information about the SB flag for this operation, see the DC_CTRL.SB bit description.
- **LM = 1 (Enable Flush on Locked Entry)** — A system-memory address written to the DC_FLDL register initiates a search for that entry. Regardless of the lock status, the DC_FLDL command generates a flush sequence if that entry is dirty. For information about the SB flag for this operation, see the DC_CTRL.SB bit description.

17.1.7 Data-Cache Region Start Address, DC_STARTR

Address: 0x4D
 Access: W
 Reset: 0x0000_0000

Figure 17-7 DC_STARTR Register



When a system memory address is written to this register, the data-cache region operation is initiated on the cache region starting with the cache line that maps to the given address (DC_STARTR), and the end address previously written to the DC_ENDR (end region) address. The address specified by DC_ENDR is excluded.

The operation performed on the cache region is defined by the DC_CTRL.RGN_OP field. You must set the DC_CTRL and DC_ENDR registers before writing to the DC_STARTR register.

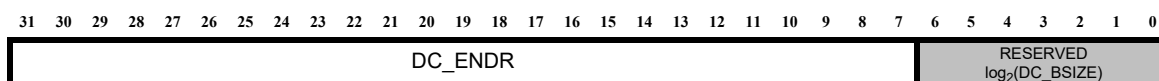
This register is write-only, and writing to this register triggers the region operation. For information about the SB flag for this operation, see the DC_CTRL.SB bit description.

When the processors is configured with MMU and the physical address extension is enabled, the build includes the DC_PTAG_HI register to hold the upper bits [39:32] of the physical address. The DC_PTAG_HI register provides the physical page number for both start and end of region. Therefore a region operation cannot span a 4G boundary.

17.1.8 Data-Cache Region End Address, DC_ENDR

Address: 0x4E
 Access: RW
 Reset: 0x0000 0000

Figure 17-8 DC_ENDR Register



This register is used to hold the last memory address for data-cache region operations. A region operation, is performed on the cache region starting from DC_STARTR to DC_ENDR, excluding the line specified by DC_ENDR. The specific region operations to be performed are specified in by the RGN_OP field in the DC_CTRL register.

You must set this register and DC_CTRL before writing to the DC_STARTR register, which triggers the operation. The SB flag in DC_CTRL can be checked to verify the completion of an DC_STARTR operation.

The address specified should be aligned according to the cache line length, as the lower order bits in the address value are ignored. This has the effect of rounding down non-aligned addresses to the nearest cache line length boundary.

When the processors is configured with MMU and the physical address extension is enabled, the build includes the DC_PTAG_HI register to hold the upper bits [39:32] of the physical address. The DC_PTAG_HI register provides the physical page number for both start and end of region. Therefore a region operation cannot span a 4G boundary.

17.1.9 Data Cache External Access Address, DC_RAM_ADDR

Address: 0x58

Access: RW

Reset: 0x0000 0000

Figure 17-9 DC_RAM_ADDR for Cache Controlled RAM Access (AT==1)

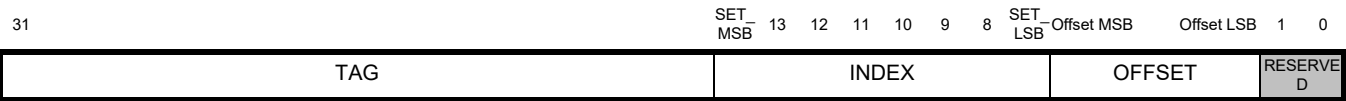
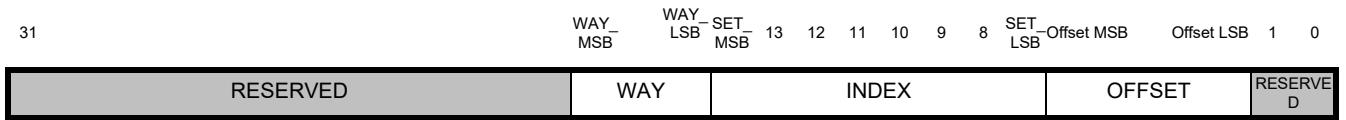


Figure 17-10 DC_RAM_ADDR Register for Direct-Cache-Access (AT==0)



This register points to a location within the cache, to be used in tandem with the DC_TAG and DC_DATA registers (refer to subsections “Data Cache Data Access, DC_DATA” and “Data Cache Tag Access, DC_TAG”).

The interpretation of the register is defined by the AT (Address Debug Type) flag in the DC_CTRL register. Writing a 0 to the AT flag in DC_CTRL selects direct cache-RAM access. Writing a 1 to the AT flag selects cache-controlled RAM access.

A detailed description of the AT flag is provided in “Data Cache Control Register, DC_CTRL” on page 964. An SR to the DC_RAM_ADDRESS register triggers a read from the data cache and updates the DC_TAG and DC_DATA registers. The read is performed according to the AT bit in the DC_CTRL register.

When the physical address extension is enabled in the MMU, the DC_PTAG_HI register is also updated with the upper bits[39:32] of the tag.

Table 17-5 DC_RAM_ADDRESS Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the MMU page size)
Direct Mode (AT==0)	■ DC_PTAG not used ■ SR to DC_RAM_ADDR: - triggers a read from RAM - WAY/INDEX/OFFSET specified by DC_RAM_ADDR - updates DC_TAG with the physical tag, lock bit and valid bit from the tag RAM, but the index field of DC_TAG is used. DC_DATA is updated with the data value from the data cache data RAM ■ SR to DC_TAG updates the DC_TAG register and has the side-effect of writing the new value to the tag RAM at the location specified by the way and index field in DC_RAM_ADDR. ■ SR to DC_DATA updates the DC_DATA register and has the side-effect of writing to the new value to the data cache data RAM at the location specified by way, index and offset in DC_RAM_ADDR. ■ LR from DC_RAM_ADDR has no side-effect and only returns the value in that register. ■ LR from DC_TAG and DC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs.	Same as non-aliased except the index field of DC_TAG is not used (remains 0).

Table 17-5 DC_RAM_ADDRESS Register Behavior

RAM Access Mode	Non-aliased	Aliased (cache way size is larger than the MMU page size)
Cache Controlled Mode (AT==1)	■ SR to DC_RAM_ADDR: - triggers a read from RAMs similar to a normal fetch access searching for data at a given address; in this case, the physical address is specified in the DC_RAM_ADDR - Selects set and way matching the given address - updates DC_CTRL.SB with cache hit status - updates DC_TAG with the physical tag, lock bit and valid bit from the tag RAM. DC_DATA is updated with the data value from data RAM; the index field of DC_TAG is updated with a copy from DC_RAM_ADDR. ■ SR to DC_TAG or DC_DATA triggers an instruction error exception. ■ LR from DC_TAG and DC_DATA only retrieves the values in those registers, but does not trigger an update of these registers with values from the tag and data RAMs.	■ SR to DC_RAM_ADDR: - The DC_PTAG register must be programmed first with the physical page number. - Then the DC_RAM_ADDR must be programmed with the virtual index and offset; this triggers a read from RAMs similar to a normal fetch access searching for data at a given address but tag matching uses the full PPN - updates DC_CTRL.SB with the cache hit status - If cache hit: DC_TAG gets the physical tag, lock and valid bits, but not the index. DC_DATA gets the data word from the data cache data RAM. DC_PTAG is unchanged ■ SR to DC_TAG or DC_DATA triggers an instruction error exception ■ LR from DC_TAG and DC_DATA only retrieves the values in those registers, but doesn't trigger an update of these registers with values from the tag and data RAMs

17.1.10 Data Cache Tag Access, DC_TAG

Address: 0x59

Access: RW

Reset: 0x0000_0000

Figure 17-11 DC_TAG for Reads



Figure 17-12 DC_TAG for Writes



Note

Following are the definitions of the various fields in this register:

- SET_LSB = $\text{Log}_2(\text{DC\_BSIZE})$
- SET_MSB = $\text{Log}_2(\text{DC\_SIZE}/\text{DC\_WAYS}) - 1$
- TAG_LSB = $\text{Log}_2(\text{DC\_SIZE}/\text{DC\_WAYS})$
- TAG_MSB = $\text{ADDR\_SIZE} - 1$

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.

When the physical address extension is enabled in the MMU, the DC_PTAG_HI register is also updated with the upper bits[39:32] of the tag.

Table 17-6 DC_TAG Control Flags and Update Conditions

Flag Name	Bit	Description	Flag Update Conditions	Access Type
V	[0]	Valid Flag: Valid flag associated with the cache line ■ 0 – Cache line not valid ■ 1 – Cache line valid	Programmer updates to DC_TAG DC_IVDL command DC_IVDC command Cache line replacement	R/W

Table 17-6 DC_TAG Control Flags and Update Conditions (Continued)

LCK	[1]	Lock Flag: Lock flag associated with the cache line ■ 0 – Cache line not locked ■ 1 – Cache line locked	Programmer updates to DC_TAG DC_LDL command DC_IVDL command DC_IVDC command	R/W
DT	[2]	Dirty Flag: Dirty flag associated with the cache line ■ 0 – Cache line not dirty ■ 1 – Cache line dirty	Programmer updates to DC_TAG DC_FLDL command DC_FLSH command DC_IVDL command DC_IVDC command Cache line replacement	R/W
EX	[3]	Exclusive flag associated with the cache line (only available in SMP configurations) 0: cache line is not exclusive in this data cache 1: cache line is exclusive in this data cache	Programmer updates to DC_TAG DC_IVDL DC_IVDC Cache line replacement	R/W

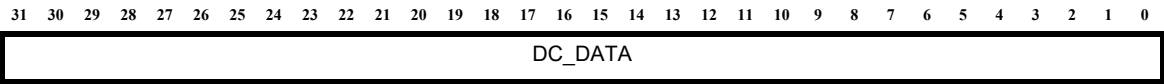
17.1.11 Data Cache Data Access, DC_DATA

Address: 0x5B

Access: RW

Reset: 0x0000_0000

Figure 17-13 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.

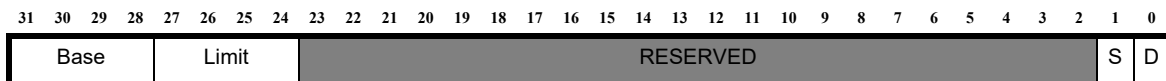
17.1.12 AUX_VOLATILE

Address: 0x5E

Access: RW

Reset: Build-time configurable

Figure 17-14 AUX_VOLATILE Register



In a single-core processor configuration, this register exists in the core auxiliary space. In a multi-core processor implementation, this register exists in the cluster auxiliary space and is shared by all the cores.

Use this register to define a volatile uncached space in the memory space. Read accesses to the volatile space are guaranteed to go out on the system bus, that is, there is never a store-to-load forwarding for the volatile space of the external memory. Write accesses to consecutive addresses in a volatile space are never merged into a single write.



Note

You can also use the MPU to define the volatile attributes of a region. For more information about what happens when both MPU and the AUX_VOLATILE define volatile attributes is discussed in the ARCV2-based processor databook.

In the non-volatile space, the read and write accesses may be optimized by the hardware. Reads may have its data forwarded from a pending store to the same address and writes to consecutive addresses may be merged.

Accesses to the peripheral region may be inside or outside the volatile space. Peripheral accesses in the volatile space are never optimized, that is, store to load forwarding and write merging are disabled. These optimizations are allowed only if the peripheral region is programmed as non-volatile.

External transactions to the volatile region can be programmed in the AUX_VOLATILE register to have a strict ordering requirement. Strict ordering means that the core serializes read and writes transactions. A volatile write transaction on any external interface, for example, the memory or peripheral interfaces is only initiated after all the previous external read transactions finish. Likewise, a volatile read transaction on any external interface is only initiated after all the previous external write transactions finish.

Strict ordering does not serialize read transactions. Read transactions from the same external interface always complete in order. It is possible that the external read transactions from different interfaces, for example: the memory or peripheral interfaces, complete out-of-order.

When the volatile transactions are programmed to not have a strict ordering requirement, the core does not serialize read and write transactions. These transactions always appear in order on the external bus. However, depending on the bus protocol, these transactions may complete out-of-order.

The Volatile Memory region defined by the `AUX_VOLATILE` register is considered a distinct memory target and accesses spanning the volatile and non-volatile memory boundary raise the exception, Memory error on data access: data access spanning multiple data memory targets.

Memory Error Exception, Access Spanning Multiple Memory Targets

A memory error exception is raised when an instruction-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



Note

- CCM memory regions are never volatile. Data access to the CCM region always go to the CCM.
- An external read followed by an external write, or an external write followed by an external read targeting the same 16-byte aligned address is always serialized by the processor, independent of the value in the `S` bit.
- An unaligned data access spanning from volatile to non-volatile (or vice-versa) region generates a memory error exception.
- Avoid programming volatile base to a value greater than or equal to the volatile limit; this setting makes the region from the volatile base to the highest memory address volatile.

Figure 17-15 Volatile Space

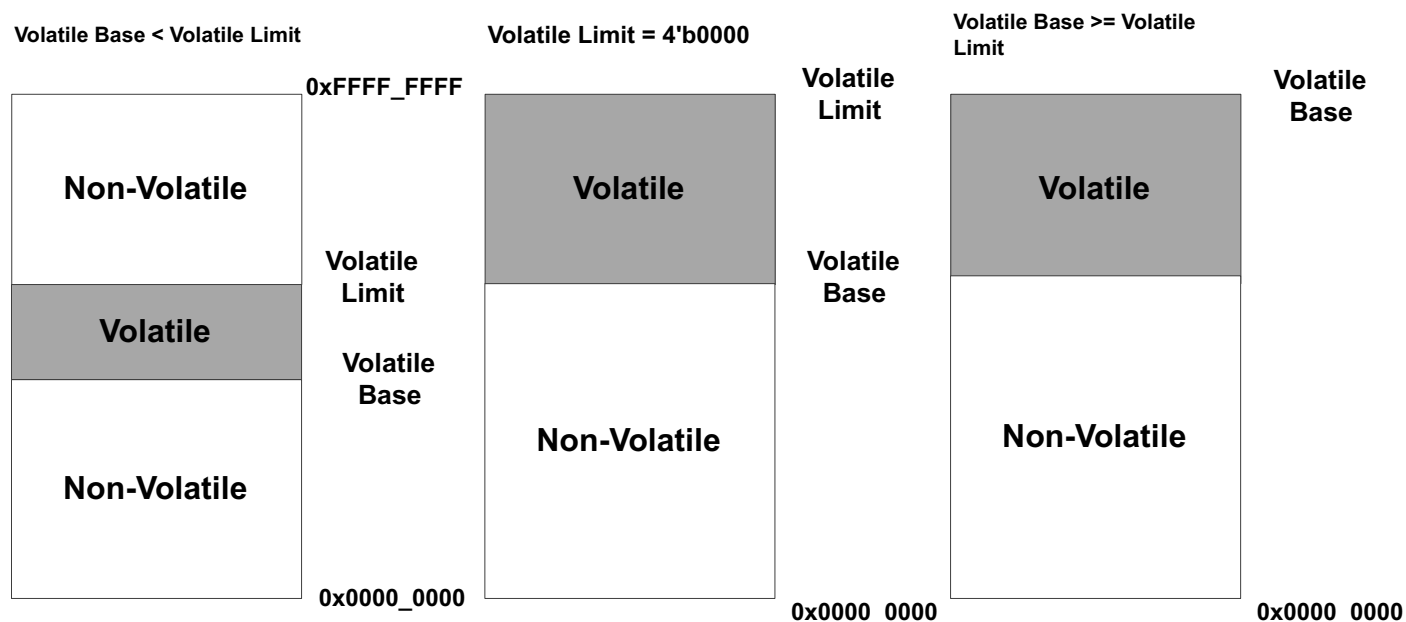


Table 17-7 AUX_VOLATILE Field Description

Field	Bit	Description
D	[0]	Disable volatile uncached space. 0x1: The entire memory space is non-volatile. 0x0: The memory space from Limit [27:24] to Base [31:28] is treated as a volatile uncached space.
S	[1]	Specifies ordering of transactions in the volatile uncached space. 0x1: Strict ordering. Reads and writes to the volatile space are serialized by the core. 0x0: Relaxed ordering. Reads and writes to the volatile space are not serialized.
Limit	[27:24]	Specifies the end of the volatile space in the memory. The volatile space excludes this limit address. Programming the volatile limit to zero means that the region from the volatile base to the highest memory address is volatile.
Base	[31:28]	Specifies the start of the volatile space in the memory. The volatile space includes the base address. When the volatile base value is greater or equal to volatile limit, the region from volatile base to the highest memory address is volatile.

17.1.13 Data Cache External Access RAM Address Physical Tag Format, DC_PTAG_HI

Address: 0x5F

Access: RW

Reset: 0x0000_0000

Figure 17-16 DC_PTAG_HI



When the ARCV2-based processor has a data cache and MMU, and the data cache way size (depth) is larger than the smallest page size, VIPT cache aliasing is avoided by cache color predication. This is implemented in the hardware to be transparent to the software and the cache is effectively PIPT. Thus, the data cache is not subject to VIPT aliasing and does not implement a DC_PTAG register.

When the physical address extension is enabled, regardless of cache or MMU page sizes, the DC_PTAG_HI register exists and provides physical address bits above bit 31. This is true for line operations as well as region operations and serves tag matching as well as providing the upper address bits for external bus access; for region operations, since both region start and region end addresses use this register for upper address bits (39:32), a region cannot span a 4G address boundary.

When the physical address extension is enabled, the ARCV2-based processor data cache (PIPT) maintenance operations require the physical tag to be written into DC_PTAG_HI register, before the virtual address (index) is written into the respective DC_* aux register.

17.2 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 17-8 Build Configuration Registers

Number	Name	Access	Description
0x72	Data Cache Configuration Register, D_CACHE_BUILD	R	Build configuration for: data cache

17.2.1 Data Cache Configuration Register, D_CACHE_BUILD

Address: 0x72

Access: R

Figure 17-17 D_CACHE_BUILD

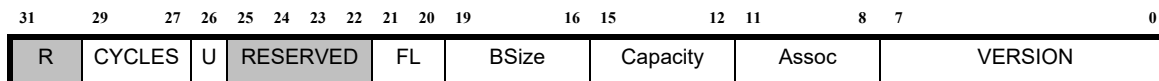


Table 17-9 D_CACHE_BUILD Field Descriptions

Field Name	Bit	Description
VERSION	[7:0]	Version number. All non-enumerated values are reserved. 0x0: No D_CACHE_BUILD register 0x1: reserved 0x2: ARCompact, fixed 32-byte line size 0x3: ARCompact, variable line size 0x4: ARCV2 with fixed number of cycles 0x5: ARCV2 with configurable cycles <i>All other values are reserved</i>
Assoc	[11:8]	Cache Associativity. All non-enumerated values are reserved. 0x0: Direct-mapped (1-way set associative) 0x1: Two-way set associative 0x2: Four-way set associative 0x3: Eight-way set associative
Capacity	[15:12]	Cache capacity. All non-enumerated values are reserved. 0x2: 2 Kbytes 0x3: 4 Kbytes 0x4: 8 Kbytes 0x5: 16 Kbytes 0x6: 32 Kbytes 0x7: 64 Kbytes

Table 17-9 D_CACHE_BUILD Field Descriptions

Field Name	Bit	Description
BSize	[19:16]	Block Size, indicates the cache block size in bytes. All non-enumerated values are reserved. 0x0: 16 bytes 0x1: 32 bytes 0x2: 64 bytes 0x3: 128 bytes 0x4: 256 bytes
FL	[21:20]	Feature Level, indicates locking and debug feature level 0x0: Basic cache, supports cache flush and invalidation operations, but no locking or debug features 0x1: Lock, flush, and invalidation features are supported 0x2: Lock, flush, invalidation, region wise cache operation, and advanced debug features are supported 0x3: <i>reserved</i>
RESERVED	[25:22]	Reserved. Read-as-zero; ignored on write.
U	26	Indicates whether the data cache contains uncached regions. 0x0: Data cache does not include uncached regions. 0x1: Data cache includes uncached regions; the AUX_CACHE_LIMIT register is present and identifies the region at which the uncached memory regions begin. Note: To define volatile and uncached data memory regions, use the AUX_VOLATILE register.
CYCLES	[29:27]	0x0: Not present 0x1: single cycle memory 0x2: two-cycle memory; always includes four banks All other values are reserved. Note: This field is valid only when D_CACHE_BUILD.VERSION==0x5.
RESERVED	[31:30]	Reserved. Read-as-zero; ignored on write.

17.2.2 Exceptions

See [Table 5-4](#) on page 233.

17.3 Cache Maintenance

Cache maintenance instructions provide a mechanism for flushing or purging data from an L1 or L2 cache. The cache maintenance commands for L1 data caches do not trigger any cache flush operations in the L2 cache.

There are two types of maintenance commands – *flush* and *purge*. A flush command causes modified data to be copied back from the L1 data cache to the next level below in the hierarchy. This would typically be the L2 cache, or external memory if there is no L2 cache or if the line is not already present in L2. A purge command first performs a flush operation, and then invalidates the line in L1 so that it cannot be read without triggering a subsequent L1 cache miss.

You can specify the addresses for the flushing and purging operations in the following ways:

- **Cache-based:** flush or purge the whole of the L1 data cache, regardless of its contents.
- **Region-based:** flush or purge a region of the physical address space, applying the operation only to those lines in L1 cache that lie within the specified region.

17.3.1 Cache-based Purge and Flush Operations

When coherency transactions are involved, the cache-based and region-based purging or flushing behave differently. When the whole cache is flushed/purged, the operation applies only to the L1 data cache of the core that issued the flush/purge command.

The whole-cache flush operation is useful when preparing a core for a deep sleep state, where it may be powered down. This operation ensures that any modified data in the L1 data cache is copied to memory before the L1 cache is powered down.

17.3.2 Region-based Flush

In a region-based flush operation, each cache line in the region is effectively flushed from all L1 data caches in the cluster. This operation is implemented using the coherency protocol.

The region-based flush operation is useful when forcing explicit consistency between data and instruction spaces – perhaps after self-modifying code, or when the O/S kernel executes a process.

17.3.3 Region-Based Purge

In a region-based purge operation, each cache line in the region is effectively purged from all L1 data caches in the cluster. This operation is implemented using the coherency protocol.

In region-based purge operations, any copies of the cache lines within the specified region are not retained in the L1 data caches. The `DC_CTRL.IM` bit determines the mode of operation of cache maintenance purge instructions:

- `DC_CTRL.IM = 1`: Dirty lines are copied-back before the line is invalidated.

`DC_CTRL.IM = 0`: Dirty lines are not copied-back before the line is invalidated. The initiator of the invalidate maintenance instruction invalidates the line in its L1 data cache. The SCU ensures that all copies in all target caches are invalidated. No copy-back is performed for this line, regardless of its state in all L1 data caches.

18

Cluster Shared Memory

18.1 CSM Introduction

The Cluster Shared Memory (CSM) provides a high capacity high-throughput memory shared by all cores in the cluster. It can be used for instructions and data, cacheable and non-cacheable, and sharable and non-sharable address space. Therefore, the CSM can be accessed by each core using the memory access instructions, instruction fetch operations, and peripheral access.

18.2 CSM Register Interface

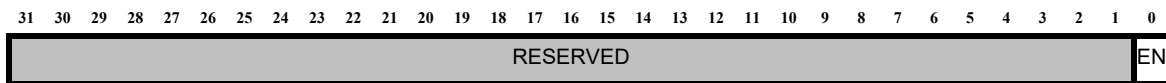
Table 18-1 CSM Registers

Address	Auxiliary Register Name	Description
0x9A0	CSM_ENABLE	Enable CSM register
0x9A1	CSM_ECC_CTRL	CSM ECC control register
0x9A2	CSM_SBE_CNT	Counter of single-bit errors that are corrected.
0x9A3	CSM_ECC_INI	Program register for initializing the CSM.
0x9A4	CSM_BASE_SYS_ADDR_L	Specifies the lower 32 bits of the CSM system base address
0x9A5	CSM_BASE_SYS_ADDR_H	Specifies the upper 8 bits of the CSM system base address when physical address extension is enabled
0xE5	CSM_BUILD	Build configuration register for CSM

18.2.1 CSM Enable Register, CSM_ENABLE

Address:	0x9A0
Additional Information:	This register is implemented in the cluster auxiliary space and belongs to the cluster power always-on domain. This register is shared by all the cores.
Access:	RW
Reset	0x1

Figure 18-1 CSM_ENABLE Register



Use this register to enable or disable CSM. This register is included in the processor build only if you have configured a multi-core cluster with CSM. Write operations to this auxiliary register are serializing, that is, a pipeline flush occurs after the auxiliary write operation commits.

You must disable CSM before powering down CSM. Similarly, after powering up CSM, you can enable CSM.

When you disable CSM, any transactions targeting CSM are not routed to CSM. Therefore, when the CSM is disabled, the effects are similar to when CSM is not configured.

Enabling or disabling CSM

Do not enable or disable CSM dynamically during runtime. If you do so, the hardware result is unpredictable. Following are some of the best practices enable or disable CSM:

1. Ensure that all the outstanding transactions accessing CSM have been completed.
 - ❑ All cores in a cluster must use the SYNC instruction or the SLEEP instruction to ensure that all the outstanding memory transactions are completed.
 - ❑ The external IOC master must stop sending transactions to IOC port, and must ensure that all the outstanding transactions are completed.
 - ❑ If a DMA component is accessing the CSM, the software must ensure that the DMA component stops accessing the CSM, and that all the outstanding transactions are completed.
2. After all the outstanding transactions accessing CSM have been completed, the software can program the CSM_ENABLE register to re-enable or re-disable CSM, and then use the SYNC instruction to ensure that the CSM_ENABLE register has been updated successfully.
3. After the CSM_ENABLE register is updated successfully, the software can re-access CSM safely.

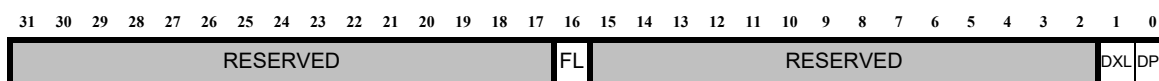
Table 18-2 CSM_ENABLE Bit Field Description

Field	Bit	Reset	Description
EN	[0]	0	Enable or disable CSM. 0x0: Disable CSM 0x1: Enable CSM (default)
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write

18.2.2 CSM ECC Control Register, CSM_ECC_CTRL

Address:	0x9A1
Additional Information:	This register is implemented in the cluster auxiliary space This register is shared by all the cores.
Access:	RW
Reset	0x0

Figure 18-2 CSM_ECC_CTRL Register



This register allows you to enable or disable ECC and parity protection on the CSM path.

Table 18-3 CSM_ECC_CTRL Bit Field Description

Field	Bit	Reset	Description
DPL	[0]	0	Control bit for ECC/parity protection for CSM; If all the CSM memory space (including partition 0, partition 1 and shared memory) ECC or parity protection is not configured, the value of the DPL bit is always 1, writes are ignored. 0x0: enables protection. On reset, the control bits are set to 0; that is ECC/parity protection is enabled. 0x1: disables protection; if protection is disabled, the protection exception is disabled regardless of the value of the DXL bit, and the FL flag is never set, ECC or parity logic is disabled and never detects and corrects errors, and the partial write does not trigger a read-modify-write sequence.
DXL	[1]	0	Control bit for enabling/disabling protection exceptions on CSM; If all the CSM memory space (including partition 0, partition 1 and shared memory) ECC or parity protection is not configured, the value of the DXL bit is always 1, writes are ignored. 0x0: enable exceptions. On reset, the control bits are set to 0; that is ECC/parity exceptions are enabled. 0x1: disable exceptions. If protection is enabled, but the protection exception is disabled, then the ECC or parity logic detects and correct errors, but memory error exception is not raised when a multi-bit uncorrectable error or a parity error is detected.
RESERVED	[15:2]	0	Reserved. Read-as-zero; ignored on write

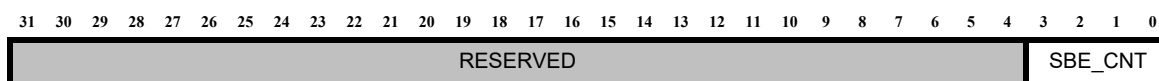
Table 18-3 CSM_ECC_CTRL Bit Field Description

FL	[16]	0	Indicates if ECC or parity errors are detected. The FL bit can only be set by the ECC hardware logic; it cannot be set by software. The software can use SR instruction to write a 0 to clear this bit. And, software can also use the LR instruction to read the FL bit. If the software writes 0 and the ECC hardware logic also sets this bit simultaneously, the software preempts the hardware. 0x0 : no errors are detected. 0x1 : double-bit error or parity error has occurred in the CSM path.
RESERVED	[31:17]	0	Reserved. Read-as-zero; ignored on write

18.2.3 CSM SBE Counter Register, CSM_SBE_CNT

Address:	0x9A2
Additional Information:	This register is implemented in the cluster auxiliary space. This register is shared by all the cores.
Access:	RW
Reset	0x0

Figure 18-3 CSM_SBE_CNT Register



This register records the number of single-bit errors that have been corrected in the CSM. This register is present only when the CSM is configured.

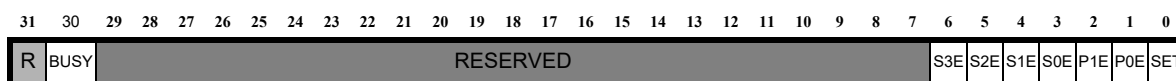
Table 18-4 CSM_SBE_CNT Bit Field Description

Field	Bit	Reset	Description
SBE_CNT	[3:0]	0	ECC single-bit correction error counter. - Software must write a 0 to clear this register. Values other than 0 must not be written. - If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. - When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. - When ECC option is not configured, this register always return 0 (zero).
RESERVED	[31:4]	0	Reserved. Read-as-zero; ignored on write

18.2.4 CSM ECC Initialization Register, CSM_ECC_INI

Address:	0x9A3
Additional Information:	This register is implemented in the cluster auxiliary space. This register is shared by all the cores.
Access:	RW
Reset	0x0

Figure 18-4 CSM_ECC_INI Register



Use this register to initialize a processor from the CSM memory. This register is present only when the CSM is configured. Otherwise, this register returns zero. If the ECC protection is not configured on all the CSM partitions and shared memory, writes to this register are ignored and reads always return 0.

Table 18-5 CSM_ECC_INI Bit Field Description

Field	Bit	Reset	Description
SET	[0]	0	Trigger CSM Parity/ECC memory initialization. 0x0: Do not trigger memory initialization 0x1: Trigger CSM initialization When read, this bit always returns 0. When the CSM is already being initialized (<code>CSM_ECC_INI[30] == 1</code>), writes to this bit are ignored.
P0E	[1]	0	Enable initialization of CSM partition 0 0x0: Disable initialization of CSM partition 0 0x1: Enable initialization of CSM partition 0
P1E	[2]	0	Enable initialization of CSM partition 1 0x0: Disable initialization of CSM partition 1 0x1: Enable initialization of CSM partition 1 Note: If CSM partition 1 is not configured, writes of 1 to this bit are ignored.
S0E	[3]	0	Enable initialization of shared memory quadrant 0 0x0: Disable initialization of shared memory quadrant 0 0x1: Enable initialization of shared memory quadrant 0 Note: If shared memory quadrant 0 is not configured or is not allocated to CSM, writes of 1 to this bit are ignored.

Table 18-5 CSM_ECC_INI Bit Field Description

S1E	[4]	0	Enable initialization of shared memory quadrant 1 0x0: Disable initialization of shared memory quadrant 1 0x1: Enable initialization of shared memory quadrant 1 Note: If shared memory quadrant 1 is not configured or is not allocated to CSM, writes of 1 to this bit are ignored.
S2E	[5]	0	Enable initialization of shared memory quadrant 2 0x0: Disable initialization of shared memory quadrant 2 0x1: Enable initialization of shared memory quadrant 2 Note: If shared memory quadrant 2 is not configured or is not allocated to CSM, writes of 1 to this bit are ignored.
S3E	[6]	0	Enable initialization of shared memory quadrant 3 0x0: Disable initialization of shared memory quadrant 3 0x1: Enable initialization of shared memory quadrant 3 Note: If shared memory quadrant 3 is not configured or is not allocated to CSM, writes of 1 to this bit are ignored.
RESERVED	[29:7]	0	Reserved. Read-as-zero; ignored on write
Busy	[30]	0	Indicates whether there is any pending initialization operation 0x0: No pending initialization operation 0x1: An initialization operation is in progress This is a read-only field. Writes to this bit are ignored.
RESERVED	[31]	0	Reserved. Read-as-zero; ignored on write

18.2.5 Initializing the CSM

If the CSM in your processor is configured you must initialize the CSM using the following procedure:

Prerequisites

- Initialize the CSM before using CSM. If the initialization is triggered when CSM is being accessed, the initialization operation waits for the current access to complete and blocks the subsequent accesses to the CSM until the initialization is complete.
- Ensure that you have already allocated the shared memory configuration as this CSM initialization sequence initializes the CSM space in a shared memory configuration.

Initialize CSM that is configured with ECC or Parity Protection

- After system is booted up or CSM is woken up from the PM2 mode, do the following:
 - Read the CSM_BUILD register to verify whether the CSM private memories are protected by ECC or parity.

- b. If you are using the processor in a shared memory configuration, read that shared memory (SLC_ECC_CONFIG register) is configured with ECC or parity protection.

If all the CSM memories are not protected by ECC or parity, process to Step 3.



If all CSM memories are powered up from the PM2 mode, the software must initialize all CSM memories that are configured with Parity/ECC. If only partial CSM memories are powered up from the PM2 mode, the software must initialize the partial powered up memories that are configured Parity/ECC.

2. Read the CSM_ECC_CTRL register to determine if ECC protection is enabled. If ECC protection is disabled, enable ECC protection by writing `CSM_ECC_CTRL[0] == 0`.



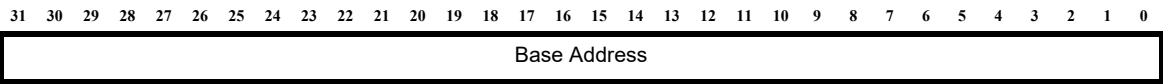
When ECC or parity protection is enabled (using register CSM_ECC_CTRL) on the CSM, the CSM is initialized with the correct parity/ECC codes. If ECC or parity protection is disabled, the initialization operation only initializes the memory data value, but does not initialize the correct parity/ECC value. The CSM data is initialized to 0 (zero).

3. Determine if an initialization sequence is already in progress: Read the CSM_ECC_INI[30] bit. If `CSM_ECC_INI[30] == 1` (that is another sequence is in progress), keep polling this bit until the status is idle (`CSM_ECC_INI[30] == 0`). When there is no pending initialization operation, initiate the initialization sequence: set `CSM_ECC_INI[0] == 1`.
4. Poll the CSM_ECC_INI[30] bit to determine if the current initialization sequence is done. After the initialization is complete, the CSM can be accessed by instructions, debugger, or DMI interface.

18.2.6 CSM System Base Lower Address Register, CSM_BASE_SYS_ADDR_L

Address:	0x9A4
Additional Information:	This register is implemented in the cluster auxiliary space. This register is shared by all the cores.
Access:	R
Reset	0x0

Figure 18-5 CSM_BASE_SYS_ADDR_L Register

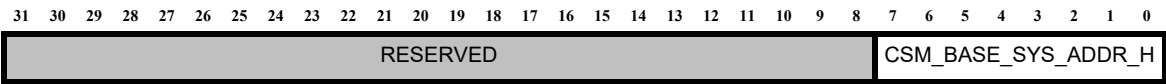


This register specifies the lower 32-bits of CSM system base address. This register reflects the value specified on the `csm_sys_addr_base [31:0]` input signal.

18.2.7 CSM System Base Higher Address Register, CSM_BASE_SYS_ADDR_H

Address:	0x9A5
Additional Information:	This register is implemented in the cluster auxiliary space. This register is shared by all the cores.
Access:	R
Reset	0x0

Figure 18-6 CSM_BASE_SYS_ADDR_H Register



This register specifies the MSB-bits of the CSM system base address. This register reflects the value specified on the `csm_sys_addr_base[39:32]` input signal.

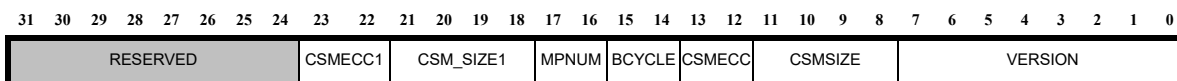
This register exists in the configuration only when CSM is configured and the physical address extension is configured (`-mmu_pae_enabled==true`) , that is, the processor supports 40-bit physical memory addresses.

18.2.8 CSM Build Configuration Register, CSM_BUILD

Address: 0xE5

Access: R

Figure 18-7 CSM_BUILD Register



This register indicates the presence of the cluster shared memory (CSM) and its configuration. If a multi-core cluster is not configured with CSM, reading this register returns zero.

If some configuration fields are invalid, reading these fields returns zero. For example, if partition number is configured as 1, CSMECC1 and CSMSIZE1 fields reading value returns zero no matter what values are configured.

Table 18-6 CSM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of CSM 0x1: initial version 0x2: support for partial power down, CSM partitioning, and SBE counter. 0x3: support for memory initialization of CSM configured with ECC and all zero and one detection.
CSMSIZE	[11:8]	CSM Size of partition 0 0x0: 32 KB 0x1: 64 KB 0x2: 128 KB 0x3: 256 KB 0x4: 512 KB 0x5: 1 MB 0x6: 2 MB 0x7: 4 MB 0x8: 8 MB 0x9 to 0xF: reserved

Table 18-6 CSM_BUILD Field Descriptions (Continued)

Field	Bit	Description
CSMECC	[13:12]	Indicates whether ECC and parity protection is enabled for CSM partition 0. 0x0: no protection. 0x1: parity protection is configured. 0x2: single-bit error correction and double-bit error detection is configured. 0x3: all zeros and ones detection is configured.
BCYCLE	[15:14]	Indicates the latency of the CSM bank. 0x0: one clock cycle 0x1: two clock cycles 0x2: three clock cycles 0x3: four clock cycles
MPNUM	[17:16]	Memory partition number 0x1: 1 memory partition 0x2: 2 memory partition
CSM_SIZE1	[21:18]	Size of CSM partition 1 0x0 : 32 KB 0x1: 64 KB 0x2: 128 KB 0x3: 256 KB 0x4: 512 KB 0x5: 1 MB 0x6: 2 MB 0x7: 4 MB 0x8: 8 MB 0x9-0xf: reserved.
CSMECC1	[23:22]	Indicates whether ECC and parity protection is enabled for CSM partition 1. 0x0: no protection. 0x1: parity protection is configured. 0x2: single-bit error correction and double-bit error detection is configured. 0x3: all zeros and ones are detection is configured.

19

L2 Cache

19.1 L2 Cache Introduction

The L2 cache is an optional component that significantly enhances the performance of multi-core cluster for large multi-threaded workloads. The L2 cache offers the following multi-processor support:

- L2-cache for ARCV2-based processors within cluster
- L2 cache size of 128K to 8 MB with configurable number of banks
- Sixteen input channels L2 cache

19.2 L2 Cache Register Interface

[Table 19-1](#) lists the L2 cache auxiliary registers. The L2 cache registers are physically located in the L2 cache, and are shared by all the cores. The L2 cache auxiliary registers are connected to a single core or through a shared auxiliary controller. For system without any core, external logic is needed to translate the request to auxiliary interface signals. If the L2 cache is instantiated in a multi-core environment, the cores must take care to ensure that only one core at a time is accessing the auxiliary registers within the L2 cache. If a core attempts to access the auxiliary registers of the L2 cache when another core is currently accessing the L2 auxiliary registers, the second request is ignored. Software should employ a semaphore to ensure exclusive access to this shared resource.

Table 19-1 L2 Auxiliary Registers

Address	Name
0x901	L2 Cache Configuration Register, SLC_AUX_CACHE_CONFIG
0x902	L2 Cache ECC Configuration Register, SLC_ECC_CONFIG
0x903	L2 Cache Control Register, SLC_AUX_CACHE_CTRL
0x904	L2 Cache Flush Register, SLC_AUX_CACHE_FLUSH
0x905	L2 Cache Invalidate Register, SLC_AUX_CACHE_INV
0x90E – 0x90F	L2 Cache Line Lock Register, SLC_AUX_LINE_LOCK
0x910 – 0x911	L2 Cache Line Invalidate Register, SLC_AUX_LINE_INV

Table 19-1 L2 Auxiliary Registers

Address	Name
0x912 – 0x913	L2 Cache Line Flush Register, SLC_AUX_LINE_FLUSH
0x914 – 0x915	L2 Cache Line Operation Start Address Register, SLC_AUX_RGN_START
0x916 – 0x917	L2 Cache Line Operation End Address Register, SLC_AUX_RGN_END
0x918 – 0x919	L2 Cache RAM Access Register, SLC_AUX_LINE_ADDR
0x91A	L2 Cache Direct Index Register, SLC_AUX_DIRECT_IDX
0x91B – 0x91C	L2 Cache TAG Register, SLC_AUX_TAG_DATA
0x91D	L2 Cache Status Register, SLC_AUX_STATUS_DATA
0x91F – 0x922	L2 Cache Data Register, SLC_AUX_DATA0 Register
0x923 – 0x924	L2 Cache Fault Address Register, SLC_AUX_FAULT_ADDR
0x925	L2 Cache Fault Status Register, SLC_AUX_FAULT_STAT
0x926	L2 Cache Performance Monitor Command Register, SLC_AUX_PM_CMD
0x927	L2 Cache Performance Monitor Event Register, SLC_AUX_PM_EVENT
0x928	L2 Cache Performance Counter Overflow Register, SLC_AUX_PM_OVF
0x929 – 0x92A	L2 Cache Performance Monitor Count0 Register, SLC_AUX_PM_CNT0
0x906 to 0x90D 0x92E to 0x936	L2 Cache Way-Lock Register, SLC_AUX_CACHE_LOCK n
0x93C	L2 Cache SBE Counter Register, SLC_AUX_ECC_SBE_CNT
0x93D	L2 Cache ECC Control Register, SLC_AUX_ECC_CTRL
0xCE	L2 Cache Build Configuration Register, SLC_BUILD

19.3 L2 Cache Configuration Register, SLC_AUX_CACHE_CONFIG

Address: 0x901

Access: R

Figure 19-1 SLC_AUX_CACHE_CONFIG 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
R	B	PM_NUM				PMS		TAG_TIME				TBANK		DATA_TIME				DBANK		WAYS		LSIZE		CACHE_SIZE							

This is a read-only register that includes the L2 cache configuration information.

The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Table 19-2 SLC_AUX_CACHE_CONFIG Field Description

Field	Bit	Description
CACHE_SIZE	[3:0]	Cache size 0x0: 128KB (default) 0x1: 256KB 0x2: 512KB 0x3: 1MB 0x4: 2MB 0x5: 4MB 0x6: 8MB 0x7 to 0xF: Reserved
LSIZE	[5:4]	Cache line size 0x0: 128Byte (default) 0x1: 64Byte 0x2 to 0x3: Reserved
WAYS	[7:6]	Ways 0x0: 4 ways (default) 0x1: 8 ways 0x2: 16 ways 0x3 to 0xF: Reserved

Table 19-2 SLC_AUX_CACHE_CONFIG Field Description

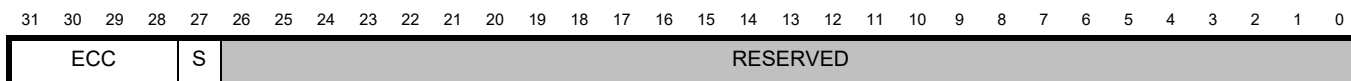
Field	Bit	Description
DBANK	[10:8]	Number of data banks 0x0: 4 banks (default for 64-byte cache line size) 0x1: 8 banks (default for 128-byte cache line size) 0x2: 16 banks 0x3: 32 banks 0x4 to 0x7: Reserved
DATA_TIME	[14:11]	Number of cycles for accessing data bank Default: 2
TBANK	[17:15]	Number of tag banks 0x0: 4 banks (default) 0x1: 8 banks 0x2: 16 banks 0x3: 32 banks 0x4 to 0x7: Reserved
TAG_TIME	[21:18]	Number of cycles for accessing the tag bank Default: 1
PMS	[23:22]	Performance monitor counter size 0x0: 32-bit (reserved) 0x1: 48-bit (default) 0x2: 64-bit (reserved) 0x3: No performance counter
PM_NUM	[28:24]	Performance counter number 0x1F: 32 counters (default)
B	[29]	Enable data-path 0x0: No data-path (default) 0x1: Enable data-path
Reserved	[31:30]	Reserved

19.4 L2 Cache ECC Configuration Register, SLC_ECC_CONFIG

Address: 0x902

Access: R

Figure 19-2 SLC_ECC_CONFIG Register



This is a read-only register that includes the L2 ECC configuration information.

The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Table 19-3 SLC_ECC_CONFIG Field Description

Field	Bit	Description
S (DATA_ECC_SIZE)	[27]	ECC size for L2 cache data memory 0x0: 32 bit 0x1: 64 bit
ECC	[31:28]	ECC and parity options 0x0: No ECC and no parity (default) 0x1: Enable parity 0x2: Enable ECC with no ECC correction writeback to memory 0x3: Enable ECC with ECC correction writeback to memory 0x4: Parity on Address and Data 0x5: ECC on Address and Data, no ECC correction writeback to memory. 0x6: ECC on Address and Data, with ECC correction writeback to memory. Table 19-11 specifies the required ECC bits when ECC protection is enabled.

Table 19-4 Data Array ECC Configuration

Data Size	ECC Encoding	Required ECC bits	Configuration Option
32 bits	32-bit encoding	7 bits	-slc_data_ecc_csize==false
64 bits	32-bit encoding	14 bits	-slc_data_ecc_csize==false
64 bits	64-bit encoding	8 bits	-slc_data_ecc_csize==true
128 bit	32-bit encoding	28 bits	-slc_data_ecc_csize==false

Table 19-4 Data Array ECC Configuration

Data Size	ECC Encoding	Required ECC bits	Configuration Option
128 bit	64-bit encoding	16 bits	-slc_data_ecc_csize==true

19.5 L2 Cache Control Register, SLC_AUX_CACHE_CTRL

Address: 0x903

Access: RW

Figure 19-3 SLC_AUX_CACHE_CTRL 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
WAY_OFF				STRIDE_OFFSET								DLF		ILF		RGN_OP		BS	LM	IM	AT	LFP		SB	R	CD					

The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Table 19-5 SLC_AUX_CACHE_CTRL Register Field Description

Field	Bit	Description
CD	[0]	Cache disable bit All regular store and load memory transfers from the core are treated as write-through and read-through. Note: If the input channel supports write-through request to the L2 cache, the channel bus width must be the same size as the memory (BIU) bus width. When the cache is disabled, CD==1: ■ All aux write commands except for the SLC_AUX_CACHE_CTRL register are ignored. ■ Performance counter is disabled ■ Line-fill prefetch is disabled ■ Aux command to read/write/flush/invalidate/lock/prefetch of any cache line in cache array is disabled. ■ Reading of aux registers is not disabled. Notes: ■ Before disabling the L2 cache (CD==1), ensure that there are no pending load or store requests to the L2 cache, and you flush and invalidate the L2 cache. ■ Before enabling the L2 cache (CD==0), ensure that there are no pending load or store requests to the L2 cache.

Table 19-5 SLC_AUX_CACHE_CTRL Register Field Description

Field	Bit	Description
SB	[2]	Success bit 0x0: Last cache operation failed 0x1: Last cache operation success SB is set (SB=1) only for the following aux operations: - Line lock: if the line cannot be locked because all ways in the sets that the address maps to are already locked, the SB flag is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when the line lock operation completes successfully. - invalidates line or region and flushes dirty entry: if the system memory address is present in the cache, the cache line is invalidated and unlocked; if the line is dirty and IM=1, it is flushed to the memory subsystem. Upon completion, the SB flag is set (SB=1) - flush line or region: if the system memory address is present in the cache, the SB flag is set (SB=1) upon completion regardless of the actual flushing. Cache flush does not set the SB bit.
LFP	[4:3]	Line Fill Prefetch option 0x0: Line-fill prefetch disable (default) 0x1: Line-fill prefetch for instruction is enabled 0x2 Line-fill prefetch for data is enabled 0x3: Line-fill prefetch is enabled for all fetches
AT	[5]	Address debug type – Use only in debug mode 0x0: Direct RAM access – L2 supports only this mode 0x1: Cache controlled RAM access – No support
IM	[6]	Invalidated mode for invalidate command 0x0: Invalidate only 0x1: Invalidate and flush (default)
LM	[7]	Lock mode effect on flush command 0x0: Disable flush on lock entry 0x1: Enable flush on lock entry (default) NOTE: Tag ways cannot be all locked; for read request, the data is discarded after fetched from external memory; for write request, the data is lost.

Table 19-5 SLC_AUX_CACHE_CTRL Register Field Description

Field	Bit	Description
BS	[8]	Busy status 0x0: Idle (default) 0x1: In aux operation (both region and single command) Before sending an auxiliary command, the processor must check to see if the L2 cache is busy by polling bit 8 (BS bit). A stale status BS bit is returned when an SR instruction (that has initiated the L2 cache maintenance operation) is followed by an LR instruction (to read the BS bit). This occurs as the core initiates the aux register read operation early in the pipeline (X1-stage). To ensure the correct status of BS bit, an SR instruction must be followed by the SYNC instruction before reading the BS bit. The SYNC instruction flushes the pipeline so that the stale BS bit read by LR instruction is discarded.
RGN_OP	[11:9]	Region operation 0x0: Region flush using the same rule as a single cache line flush 0x1: Region invalidate using the same rule as single cache line invalidation 0x2: Region prefetch; the purpose is preloading L2 cache; if the line exists, it is skipped. 0x3 to 0x7: Reserved
ILF	[15:12]	Number of instruction line fill prefetch; instruction prefetch is always sequential 0x0: 1 extra line fill prefetch (default) 0x1: 2 extra line fill prefetch 0x2: 3 extra line fill prefetch ... 0xF: 16 extra line fill prefetch
DLF	[19:16]	Number of data line fill prefetch; data prefetch is sequential only when STRIDE_OFFSET==1 0x0: 1 extra line fill prefetch (default) 0x1: 2 extra line fill prefetch 0x2: 3 extra line fill prefetch ... 0xF: 16 extra line fill prefetch
STRIDE_OFFSET	[27:20]	Stride offset is only used with data line fill prefetch. 0x0: Do not use this value. This value is reserved. 0x1: data prefetch is sequential (default) 0x2 to 0xFF: when prefetching data, , the offset is added to the current cache miss line address for fetching the next cache line.

Table 19-5 SLC_AUX_CACHE_CTRL Register Field Description

Field	Bit	Description
WAY_OFF	[31:28]	Indicates the power mode of each memory quadrant of the shared memory associated with the L2 cache. For a four-way associative, each bit is way associative. For 8-way associative, each bit is 2-way associative. For 16-way associative, each bit is 4-way associative. 0x0: the selected way is powered up. (default) 0x1: The selected way is powered down or in the retention mode; the corresponding bits in the channel way-lock register are set to 1 so that the way that is powered off is not selected for replacement.

**Note**

- Access to tag way when it is power-down or retention mode returns a tag miss. Since the tag arrays indicate tag miss for the power-off ways and the ways are locked. The corresponding data banks should not be selected in normal operation. In addition both tag array and data array can be in power-down or retention mode. Since tag array is small, tag array can be enabled but it is clock-gated off by the way_power_off bits.
- To properly power down/up the partial cache arrays, the L2 cache must wait for all current requests to be completed before modifying the Way_Off bits. Modifying the Way_Off bits is delayed to synchronize with the PDM and DVFS unit and L2 cache is idle.
 - If CD=1, then the Way_Off fields is written directly without any delay.

19.5.1 Way Power Down and Power On Procedures

For multi-core system, the L2 cache is partially enabled as the cores are enabled. The configuration must initialize the AUX Way-Power-Off registers to start with only partial cache powered on. The granularity of partial power-down or retention mode is $\frac{1}{4}$, $\frac{1}{2}$, or $\frac{3}{4}$ cache arrays.

To power-down or set retention mode from the power-up mode, the software must ensure that any data in the power-down ways must not be accessed. This is possible as the data is exclusively used by the corresponding core that is being power-down. Otherwise, the cache must be flushed and invalidated before the cache can be powered-down or put into the retention mode. The cache ways must be locked and the AUX Way_Off bits must be set to 1 before the cache arrays can be powered-down or put into retention mode.

Power-Down Partial Cache Arrays

Use the following steps to power down the partial cache arrays:

- Flush and invalidate the cache if needed.
- Program the SLC_AUX_CACHE_CTRL.WAY_OFF bits. The WAY_OFF bits are not modified until step 4.
- Program the SLC_P_PMODE in the PDM component to partially power down the L2 cache. PDM does not acknowledge the write until the L2 cache is idle. PDM sends an slc_idle_request to the L2 cache.

4. When `slc_idle_request` is asserted, the `WAY_OFF` bits are written, and PDM acknowledges the write to the `SLC_P_PMODE` register.
5. Power-down or lower the power supply for retention mode to the tag ways (if desired) and data banks.

Power-Up Partial Cache Arrays from the Retention Mode

Use the following steps to power up the partial cache arrays:

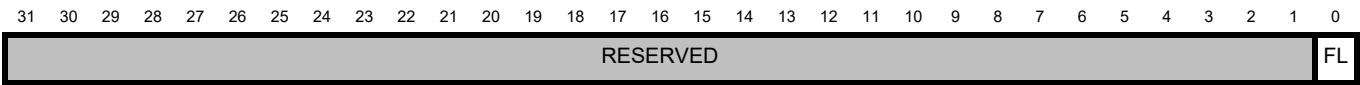
1. Program the `SLC_AUX_CACHE_CTRL.WAY_OFF` bits. The `WAY_OFF` bits are not modified until step 4.
2. Program the `SLC_P_PMODE` in the PDM component to partially power down the L2 cache. PDM does not acknowledge the write until the L2 cache is idle. PDM sends an `slc_idle_request` to the L2 cache.
3. Apply full voltage to the tag ways and data banks.
4. Program the `SLC_P_PMODE` in the PDM component to partially power down the L2 cache. PDM does not acknowledge the write until the L2 cache is idle. PDM sends an `slc_idle_request` to the L2 cache.
5. To power-up the tag and status arrays from power-down mode, the software must use cache maintenance instructions through SLC Direct Access Register to reset the tag and status arrays
 - Note: this step is not needed to power-up the arrays from retention mode.
 - Note: this step is not needed if the tag ways is not powered down.

19.6 L2 Cache Flush Register, SLC_AUX_CACHE_FLUSH

Address: 0x904

Access: W

Figure 19-4 SLC_AUX_CACHE_FLUSH Register



Write to this register to flush all the dirty cache lines in the L2 cache.

The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Table 19-6 SLC_AUX_CACHE_FLUSH Register Field Description

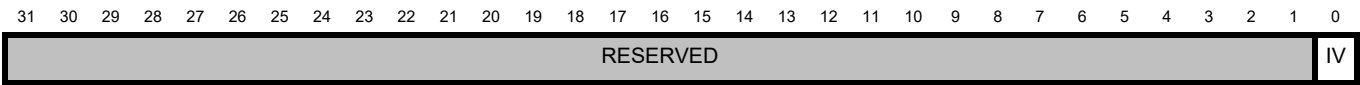
Field	Bit	Description
FL	[0]	Flush cache lines Write 1 to this register to flush all dirty cache lines in the L2 cache using the same rule as a single cache line flush.

19.7 L2 Cache Invalidate Register, SLC_AUX_CACHE_INV

Address: 0x905

Access: W

Figure 19-5 SLC_AUX_CACHE_INV Register



Write to this register to invalidate and unlock the L2 cache using the same rules as a single cache line invalidation.

The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Table 19-7 SLC_AUX_CACHE_INV Register Field Description

Field	Bit	Description
IV	[0]	Invalidate and unlock L2 cache Write 1 to this register to invalidate and unlock the L2 cache using the same rules as a single cache line flush.

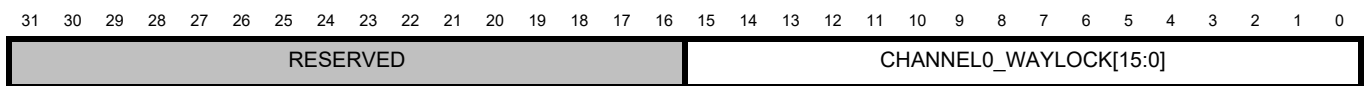
19.8 L2 Cache Way-Lock Register, SLC_AUX_CACHE_LOCK n

Address: 0x906 to 0x90D
0x92E to 0x936

Access: RW

Reset: 0x00000000

Figure 19-6 Cache Way-Lock Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

The cache way-lock register lets you lock access or share ways across cores. The number of connected ports depends on the L1 instruction and data cache configuration and whether the processor includes a shared coherency unit.

The number of way-lock registers is calculated as follows:

- When a coherency unit is present in the cluster:

$$\text{number of way-lock registers} = \text{number\_of\_cores} + 3$$
- When there is no coherency unit in the cluster:

$$\text{number of way-lock registers} = \text{number\_of\_cores} + (2 * (\text{num\_cores\_with\_data\_cache})) + 1$$

Where:

- A Coherency Unit is present in the cluster if the `CLUSTER_BUILD[NUM_ENTRIES]` field contains a non-zero value (bits [23:16]).
- `number_of_cores` is given by the `CLUSTER_BUILD[NUM_CORES]` field (bits [15:8]).
- Data cache is configured in a core if the `D_CACHE_BUILD` register of the core contains a non-zero value.

The way-lock registers for a core occupy consecutive addresses. Way-lock registers for core0 start at address 0x906. For example, for core0 with three ports, the way-lock registers are `SLC_AUX_CACHE_LOCK0`, `SLC_AUX_CACHE_LOCK1`, and `SLC_AUX_CACHE_LOCK2`.

The number of bits in each way-lock register is dependent on the number way associativity of L2 cache. For 4-way associativity, each way-lock register has only 4 bits.

Each bit [15:0] in a way-lock register (`SLC_AUX_CACHE_LOCK $n$`) corresponds to a way. A bit value of 1 indicates that the way is locked for the corresponding port; the core cannot replace that way. A bit value of 0 indicates that the port can replace that way.

Following are some of the guidelines for locking ways:

- All ways cannot be locked. If all the ways are locked:
 - for read requests the data is discarded after data is fetched from external memory.
 - for write requests, data is lost; the `err_wr` signal is asserted.
- Ways can be shared among the cores. For example, to share way0 between ports 0 and 1 and lock way0 for ports 2 and 3, do the following:
 - Set `SLC_AUX_CACHE_LOCK2 [0]` and `SLC_AUX_CACHE_LOCK3 [0]` to 1.
 - Set `SLC_AUX_CACHE_LOCK0 [0]` and `SLC_AUX_CACHE_LOCK1 [0]` to 0.
- Line fill prefetch uses the same way-lock register as the original miss request.
- The number of way-lock registers include the prefetch port; for 16 ports, there are 17 way-lock registers.
- Aux-region-prefetch and lock-line-prefetch operations use the prefetch-way-lock register.

**Note**

The last two entries of the cache with the most significant way, that is, way 3 of 4-way associative, is locked for saving and restoring of aux control registers. Setting direct-map for a port should avoid using the most significant way as this can cause all-way lock for the last two entries of the cache.

Application notes:

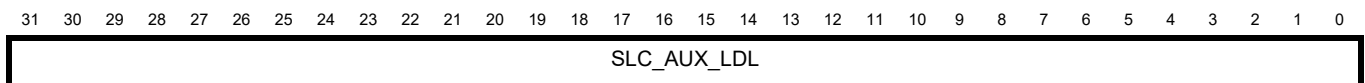
- The prefetch-port way-lock register should be programmed to be the same as the intended port way-lock register before the aux-region-prefetch or lock-line prefetch operation. The prefetched data is put into the same allocated ways as the intended port. For example, if ways 0-3 (8-way associative) are allocated for core0, the prefetched data for core0 should be loaded into ways 0-3 by setting the prefetch-way-lock register to 11110000.
- Region prefetch can be selective to load prefetch data into specific way0 such as 11111110 to prevent replacement of useful data.
- Similarly, lock-line operation fetches the cache line from external memory if the lock cache line is not present in L2 cache. If the lock-line request is from core0, then prefetch-way-lock register should be set to the same way-lock register for core0.
- The way replacement selection takes affect after the auxiliary way-lock registers are programmed. When way-lock registers are programmed, the pending requests in the L2 cache may select a different replacement way than expected. Eventually, data should be replaced by the correct channel way-lock registers.

19.9 L2 Cache Line Lock Register, SLC_AUX_LINE_LOCK

Address: 0x90E

Access: W

Figure 19-7 SLC_AUX_LINE_LOCK Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Note that for 40-bit or 64-bit address, two registers are used for storing the address. The upper-address register (0x90F) must be written before the lower-address register (0x90E). Writing to the lower-address register triggers the lock operation.

Note that the address registers for flush, invalidate, lock, read, and region operation (start address register) are write-only registers. Only one operation is allowed at a time, software must read the SB and BS bits in the L2 cache control register before starting the next operation.

When the address is written to this register, the L2 cache verifies if the request line is present in the cache:

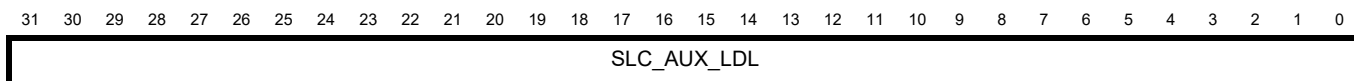
- If valid (tag-hit) and dirty
 - SLC_AUX_CACHE_CTRL.LM=1, the line is flushed to memory and locked.
 - SLC_AUX_CACHE_CTRL.LM=0, the line is locked without flushing.
- If valid and not dirty, the line is re-fetched and locked.
- If not valid, then the line is fetched and locked.
- Locking the last way is not allowed. The SB bit remains 0.

19.10 L2 Cache Line Lock High Register, SLC_AUX_LINE_LOCK1

Address: 0x90F

Access: W

Figure 19-8 SLC_AUX_LINE_LOCK Register



This register is used for storing the upper bits of a 40-bit or 64-bit address, the lower bits are stored in the lower address register (0x90E). The upper-address register (0x90F) must be written before the lower-address register (0x90E). Writing to the lower-address register triggers the lock operation.

This register exists in the build only when the physical address extension is configured (-mmu_pae_enabled==true), that is, the processor supports 40-bit physical memory addresses.

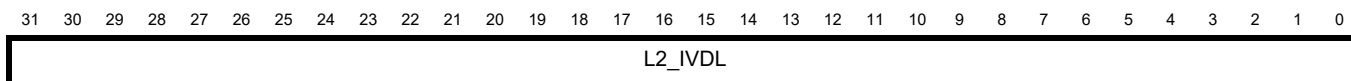
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.11 L2 Cache Line Invalidate Register, SLC_AUX_LINE_INV

Address: 0x910

Access: W

Figure 19-9 SLC_AUX_LINE_INV Register



Note that for 40-bit or 64-bit address, two registers are used for storing the address. The upper-address register (0x911) must be written before the lower-address register (0x910). Writing to the lower-address register triggers the invalidation operation.

The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

When an address is written to this register, the L2 cache verifies if the request line is present in the cache:

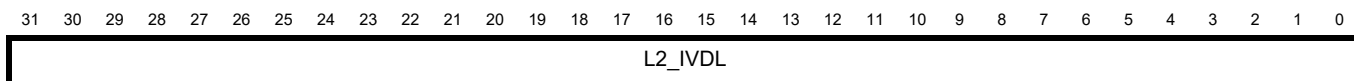
- If valid (tag hit) and dirty
 - SLC_AUX_CACHE_CTRL.IM=1, the line is flushed to memory and invalidated.
 - SLC_AUX_CACHE_CTRL.IM=0, the line is invalidated.
- If valid and not dirty, the line is invalidated.
- If not valid, then no operation.

19.12 L2 Cache Line Invalidate High Address Register, SLC_AUX_LINE_INV1

Address: 0x911

Access: W

Figure 19-10 SLC_AUX_LINE_INV1 Register



This register is used for storing the upper bits of a 40-bit or 64-bit address. The lower bits are stored in the lower address register (0x910). The upper-address register (0x911) must be written before the lower-address register (0x910). Writing to the lower-address register triggers the invalidation operation.

This register exists in the build only when the physical address extension is configured (`-mmu_pae_enabled==true`), that is, the processor supports 40-bit physical memory addresses.

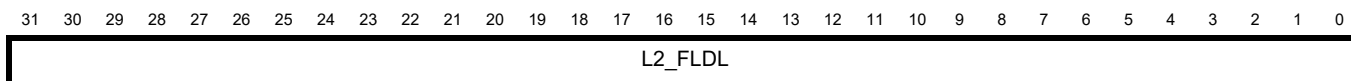
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.13 L2 Cache Line Flush Register, SLC_AUX_LINE_FLUSH

Address: 0x912

Access: W

Figure 19-11 SLC_AUX_LINE_FLUSH Register



Note that for 40-bit or 64-bit address, two registers are used for storing the address. The upper-address register (0x913) must be written before the lower-address register (0x912) is written. Writing to the lower-address register triggers the flush operation.

The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

When an address is written to this register, the L2 cache verifies if the request line is present in the cache:

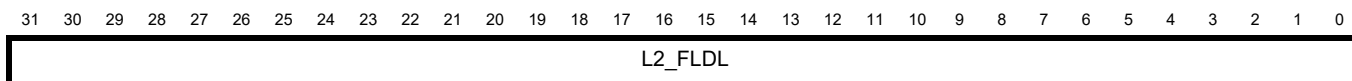
- If valid (tag hit) and dirty
 - SLC_AUX_CACHE_CTRL.LM=1, the line is flushed to memory regardless of the lock status.
 - SLC_AUX_CACHE_CTRL.LM=0, if the line is locked, no operation.
- If valid and not dirty, no operation is triggered.
- If not valid, then no operation is triggered.

19.14 L2 Cache Line Flush High Address Register, SLC_AUX_LINE_FLUSH1

Address: 0x913

Access: W

Figure 19-12 SLC_AUX_LINE_FLUSH1 Register



This register is used for storing the upper bits of a 40-bit or 64-bit address. The lower bits are stored in the lower address register (0x912). The upper-address register (0x913) must be written before the lower-address register (0x912). Writing to the lower-address register triggers the lock operation.

This register exists in the build only when the physical address extension is configured (`-mmu_pae_enabled==true`), that is, the processor supports 40-bit physical memory addresses.

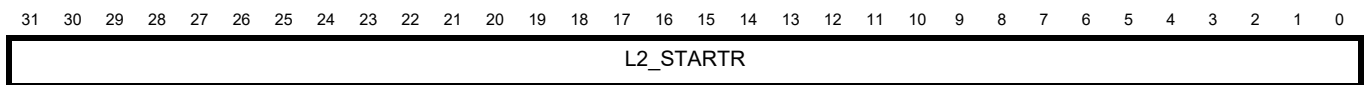
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.15 L2 Cache Line Operation Start Address Register, SLC_AUX_RGN_START

Address: 0x914

Access: W

Figure 19-13 SLC_AUX_RGN_START Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

The region operation includes two address registers, the start address and the end address registers. The end address registers (SLC_AUX_RGN_END and SLC_AUX_RGN_END1) and the SLC_AUX_RGN_START1 register must be written before the SLC_AUX_RGN_START register. Writing to the SLC_AUX_RGN_START register triggers the region operation.

For cache line size = 128 Bytes, address bits [6:0] are ignored.

For cache line size = 64 Bytes, address bits [5:0] are ignored.

The operation is based on the SLC_AUX_CACHE_CTRL.RGN_OP[2:0] bits.

- 000: flush operation
 - SLC_AUX_CACHE_CTRL.LM=1, the line is flushed to memory regardless of the lock status.
 - SLC_AUX_CACHE_CTRL.LM=0, no operation if the line is locked.
- 001: invalidate operation
 - SLC_AUX_CACHE_CTRL.IM=1, the line is flushed to memory and invalidated.
 - SLC_AUX_CACHE_CTRL.IM=0, the line is invalidated.
- 010: prefetch operation
 - If the cache line is present in the cache, then no operation.
 - If the cache line is not present in the cache, then it is fetched from external memory.

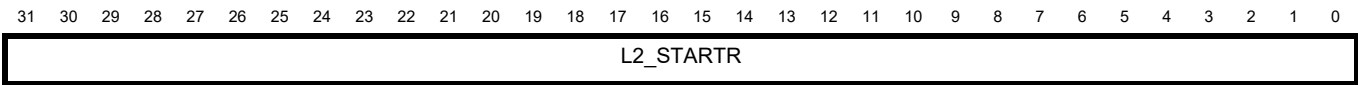
19.16

L2 Cache Line Operation Start High Address Register,
SLC_AUX_RGN_START1

Address: 0x915

Access: W

Figure 19-14 SLC_AUX_RGN_START1 Register



This register is used for storing the upper bits of a 40-bit or 64-bit address. The end address register (SLC_AUX_RGN_END) must be written before the SLC_AUX_RGN_START register.

This register exists in the build only when the physical address extension is configured (-mmu_pae_enabled==true), that is, the processor supports 40-bit physical memory addresses.

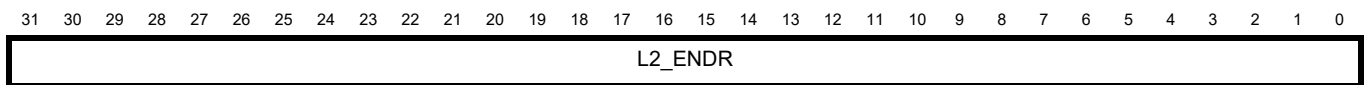
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.17 L2 Cache Line Operation End Address Register, SLC_AUX_RGN_END

Address: 0x916

Access: W

Figure 19-15 SLC_AUX_RGN_END Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

The region operation includes two address registers, the start address and the end address registers. The end address register must be written before the SLC_AUX_RGN_START register.

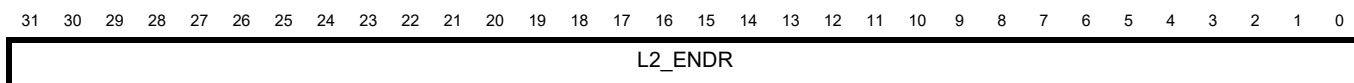
Note that the end address is excluded from the region operation. The end address cannot be the same as the start address. Setting the end address the same as the start address causes unpredictable behavior; all addressable memory are cycled through.

19.18 L2 Cache Line Operation End High Address Register, SLC_AUX_RGN_END1

Address: 0x917

Access: W

Figure 19-16 SLC_AUX_RGN_END1 Register



This register is used for storing the upper bits of a 40-bit or 64-bit region end address. The end address register must be written before the SLC_AUX_RGN_START register .

This register exists in the build only when the physical address extension is configured (-mmu_pae_enabled==true) , that is, the processor supports 40-bit physical memory addresses.

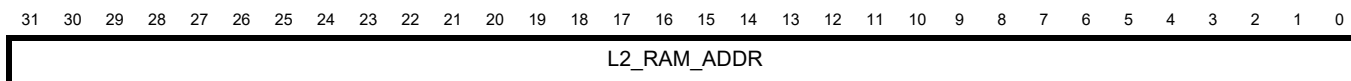
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.19 L2 Cache RAM Access Register, SLC_AUX_LINE_ADDR

Address: 0x918

Access: W

Figure 19-17 SLC_AUX_LINE_ADDR Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Note that for 40-bit or 64-bit address, two registers are used for storing the address. The upper-address register (0x919) must be written before the lower-address register (0x918) is written. Writing to the lower-address register triggers RAM read access operation.

The RAM access address register is used to check for tag hit in the tag array. The information is returned in the following registers.

- SLC_AUX_TAG_DATA register: valid and way indicate the present of the cache line address in the array.
- SLC_AUX_STATUS_DATA register: the lock, dirty, and LRU bits are also read.

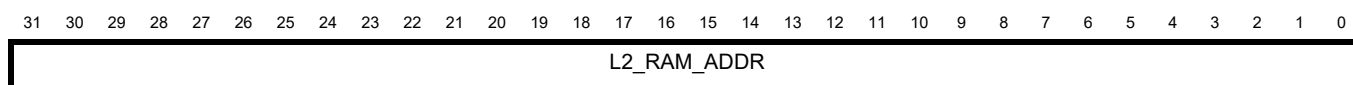
No other operation is implemented for this register.

19.20 L2 Cache Line RAM Access High Address Register, SLC_AUX_LINE_ADDR1

Address: 0x919

Access: W

Figure 19-18 SLC_AUX_LINE_ADDR1 Register



This register stores the upper bits of the 40-bit or 64-bit address. The upper-address register (0x919) must be written before the lower-address register (0x918) is written. Writing to the lower-address register (0x918) triggers the RAM read access operation.

This register exists in the build only when the physical address extension is configured (-mmu_pae_enabled==true), that is, the processor supports 40-bit physical memory addresses.

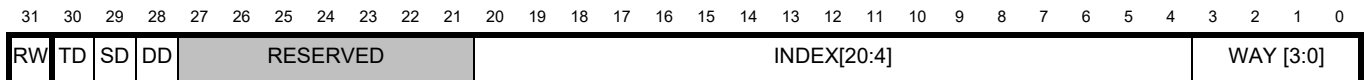
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.21 L2 Cache Direct Index Register, SLC_AUX_DIRECT_IDX

Address: 0x91A

Access: W

Figure 19-19 SLC_AUX_DIRECT_IDX Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Writing to this register triggers a read or write operation to the tag, data, and status data arrays. Based on a combination of the disable bits [30:28] and the read or write mode bit [31], the SRAM arrays are read or written to the SLC_AUX_TAG_DATA, SLC_AUX_DATA_n, and SLC_AUX_STATUS_DATA registers. You can read or write simultaneously to the tag, data, or status registers. For more details, see [Table 19-8](#). Note that with direct access, you cannot modify the reserved cache line (last 1 or 2 cache lines with last way associativity) for saving the auxiliary registers in the deep sleep mode. An attempt to write to these cache lines is ignored.



Note

After initiating a read or write command, you must poll the SLC_AUX_CACHE_CTRL.BS bit. If the BS bit is 1, the L2Cache is busy in handling the initiated operation. Ensure that the BS bit is 0 before reading from the SLC_AUX_TAG_DATA, SLC_AUX_DATA_n, and SLC_AUX_STATUS_DATA registers. Also after a write operation, ensure that the BS bit is 0 before initiating another command.

Table 19-8 SLC_AUX_DIRECT_IDX Field Description

Field	Bit	Description
WAY	[3:0]	Selected way; up to 16-way associative
INDEX	[20:4]	Index to arrays, not all bits are used for cache size smaller than 8 MB - Tag_Index[20:7] for 128-byte cache line size, Tag_Index[20:6] for 64-byte cache line size - Status_Index[20:7] for 128-byte cache line size, Status_Index[20:6] for 64-byte cache line size - Data_Index[20:4] to index to data banks. Note: Tag_index[20] is based on the maximum size of 8 MB = 23-bit index, the minimum number of ways is 4 = 2bit, so 21-bit index is the maximum bits needed for index.

Table 19-8 SLC_AUX_DIRECT_IDX Field Description

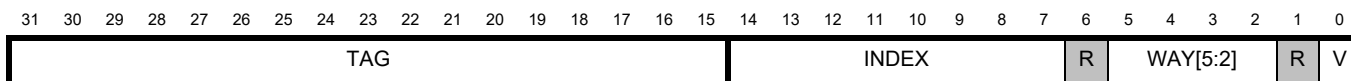
Field	Bit	Description
DD	[28]	DD – Data disable 0x0: read/write data array 0x1: do not read/write data array
SD	[29]	SD – Status disable 0x0: read/write status array 0x1: do not read/write status array
TD	[30]	TD – Tag disable 0x0: read/write tag array 0x1: do not read/write tag array
RW	[31]	RW – Read or write to arrays 0x0: Write the SLC_AUX_TAG_DATA, SLC_AUX_DATA_n, and SLC_AUX_STATUS_DATA, registers to SRAM arrays 0x1: Read SRAM arrays to the SLC_AUX_TAG_DATA, SLC_AUX_DATA_n, and SLC_AUX_STATUS_DATA registers

19.22 L2 Cache TAG Register, SLC_AUX_TAG_DATA

Address: 0x91B

Access: RW

Figure 19-20 SLC_AUX_TAG_DATA Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Note that for 40-bit or 64-bit addresses, two registers are used. The upper-address SLC_AUX_TAG_DATA1 register contains the upper bits of the tag address.

Based on a combination of the disable bits SLC_AUX_DIRECT_IDX [30] and the read/write mode bit SLC_AUX_DIRECT_IDX [31], the SRAM array is read or written to the SLC_AUX_TAG_DATA register. For more details, see [Table 19-8](#).

Table 19-9 SLC_AUX_TAG_DATA Register Field Description

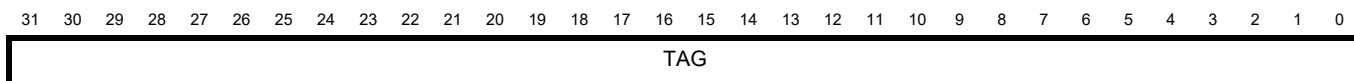
Field	Bit	Description
V	[0]	From tag array
WAY	[5:2]	Selected way, up to 16-way associative, for 4-way associative only bits [3:2] are used
R	[6]	Reserved in 128-byte cache line size For 64-byte cache line size, this bit is part of the index.
INDEX[index_range]	[14:7]	The index to tag banks
TAG[tag_range]	[31:15]	Tag address from tag banks

19.23 L2 Cache TAG High Address Register, SLC_AUX_TAG_DATA1

Address: 0x91C

Access: RW

Figure 19-21 SLC_AUX_TAG_DATA1 Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

This register contains the upper 8-bits of the tag address in its LSB. For 64-bit address, this register stores the upper bits of the tag address. The lower bits of the tag address are stored in the SLC_AUX_TAG_DATA register (0x91B).

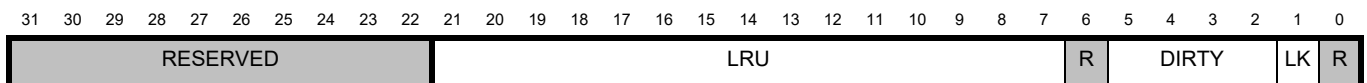
This register exists in the build only when the physical address extension is configured (-mmu_pae_enabled==true), that is, the processor supports 40-bit physical memory addresses.

19.24 L2 Cache Status Register, SLC_AUX_STATUS_DATA

Address: 0x91D

Access: RW

Figure 19-22 SLC_AUX_STATUS_DATA Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Based on the combination of the disable bits `SLC_AUX_DIRECT_IDX` [29] and the read/write mode bit `SLC_AUX_DIRECT_IDX` [31], the SRAM array is read or written to the `SLC_AUX_STATUS_DATA` register. For more details, see [Table 19-8](#).

Table 19-10 SLC_AUX_STATUS_DATA Register Field Description

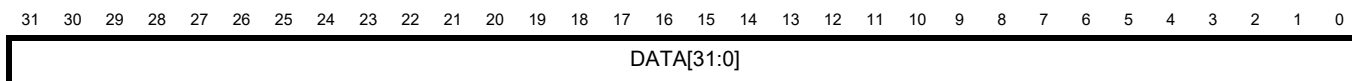
Field	Bit	Description
LK	[1]	Lock bit from status array
DIRTY	[5:2]	Dirty bits from status array, for 64-byte cache line size, only bits [3:2] are used.
LRU [LRU_range]	[21:7]	The LRU of the current tag index, 7 bits for 8-way associative, more or fewer bits for other associativity ■ [9:7]: 3-bit LRU for 4-way associative ■ [13:7]: 7-bit LRU for 8-way associative ■ [21:7]: 15-bit LRU for 16-way associative

19.25 L2 Cache Data Register, SLC_AUX_DATA0 Register

Address: 0x91F

Access: RW

Figure 19-23 SLC_AUX_DATA0 Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Each data bank is 128-bit wide; all output data from a single data bank can be stored in these registers. For writing of data into data bank, width is always 128-bit. The registers at addresses from 0x91F to 0x922 are used to store this data. For writing of 32-bit or 64-bit, the cache line should be read into the SLC_AUX_DATA n registers and modified to the appropriate 32 or 64-bit before writing back to data bank.

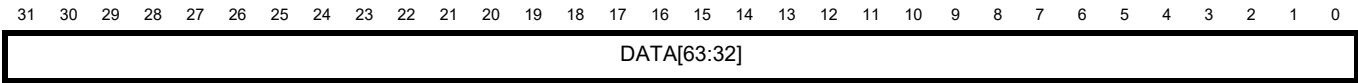
The RAM access operation uses the SLC_AUX_TAG_DATA, SLC_AUX_STATUS_DATA, and SLC_AUX_DATA n registers for reading and writing of data.

19.26 L2 Cache Data High Register, SLC_AUX_DATA1

Address: 0x920

Access: RW

Figure 19-24 SLC_AUX_DATA1 Register



Each data bank is 128-bit wide; all output data from a single data bank can be stored in these registers. For writing of data into data bank, width is always 128-bit. The registers at addresses from 0x91F to 0x922 are used to store this data.

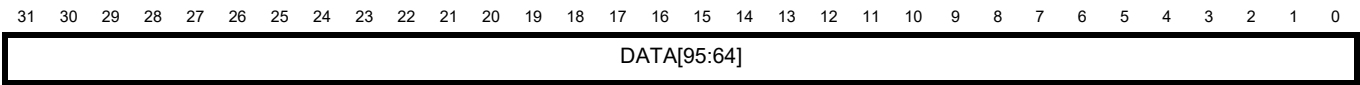
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.27 L2 Cache Data2 Register, SLC_AUX_DATA2

Address: 0x921

Access: RW

Figure 19-25 SLC_AUX_DATA2 Register



Each data bank is 128-bit wide; all output data from a single data bank can be stored in these registers. For writing of data into data bank, width is always 128-bit. The registers at addresses from 0x91F to 0x922 are used to store this data.

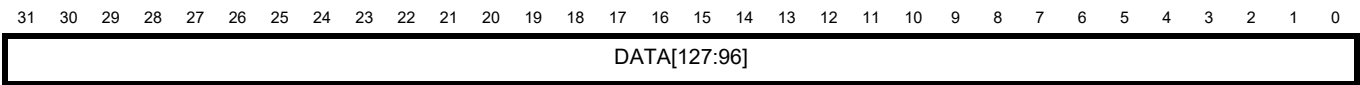
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.28 L2 Cache Data3 Register, SLC_AUX_DATA3

Address: 0x922

Access: RW

Figure 19-26 SLC_AUX_DATA3 Register



Each data bank is 128-bit wide; all output data from a single data bank can be stored in these registers. For writing of data into data bank, width is always 128-bit. The registers at addresses from 0x91F to 0x922 are used to store this data.

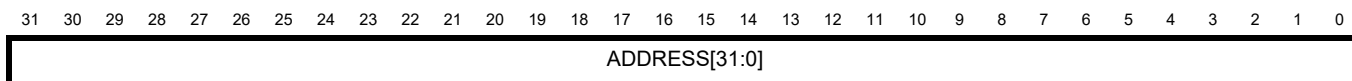
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.29 L2 Cache Fault Address Register, SLC_AUX_FAULT_ADDR

Address: 0x923

Access: R

Figure 19-27 SLC_AUX_FAULT_ADDR Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

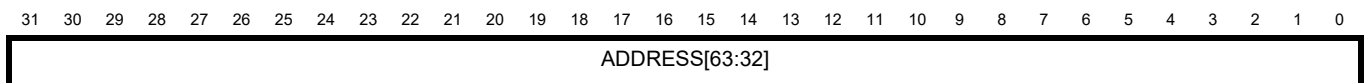
This register contains the lower 32 bits the address with error. Until the registered fault status bits in the SLC_AUX_FAULT_STAT register are cleared, only the oldest error is captured; subsequent errors are not captured. For 40-bit or 64-bit address, two fault address registers are used.

19.30 L2 Cache Fault Address High Register, SLC_AUX_FAULT_ADDR1

Address: 0x924

Access: R

Figure 19-28 SLC_AUX_FAULT_ADDR1 Register



This register contains the upper 32 bits of the address with error. Until the registered fault status bits in the SLC_AUX_FAULT_STAT register are cleared, only the oldest error is captured; subsequent errors are not captured. This register is used only for 40-bit or 64-bit addresses.

This register exists in the build only when the physical address extension is configured (-mmu_pae_enabled==true), that is, the processor supports 40-bit physical memory addresses.

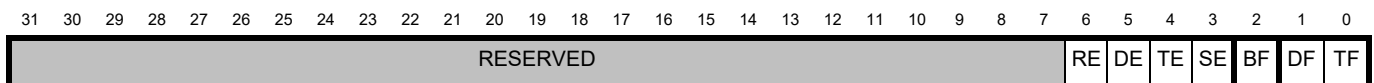
The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

19.31 L2 Cache Fault Status Register, SLC_AUX_FAULT_STAT

Address: 0x925

Access: RW

Figure 19-29 SLC_AUX_FAULT_STAT REGISTER



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

This register records the ECC/parity faults that occur during data eviction and auxiliary operation.

Two imprecise errors are defined:

- Data eviction error: this error asserts interrupt signal to the core and sets up the fault address register as well as error bits in this fault status register. The error can be from the following sources:
 - Tag array ECC/parity error
 - Data array ECC/parity error
 - Memory bus error
- Aux operation error – this error asserts the interrupt signal to the core and set up the fault address register and error bits in this fault status register. The error can be from four different sources:
 - Status array ECC/parity error
 - Tag array ECC/parity error
 - Data array ECC/parity error
 - ECC/parity error from restoring of aux registers after deep sleep mode. The aux registers remain in the reset state and are not updated in case of an error. Also, the fault address register is not updated as data is restored from the data array's last two entries.
- Writing of prefetch data into cache arrays. This error is from a tag miss that requires external data fetch. The request is completed and fetched data is written into the cache array. Full writes for tag and data arrays cause no possible error. The status array requires read-modify-write operations that can have ECC error; in this case, the fetched tag and data is written into the arrays but not the status array. The next cache access to this cache index sees the error. This error causes an interrupt to the core.

Table 19-11 SLC_AUX_FAULT_STAT Register Field Description

Field	Bit	Description
Copy-Back Error		
TF	[0]	0x0: No error 0x1: Tag array ECC/parity error on read - Writing 1 clears the error bit. - Writing 0 has no effect on the error bit.
DF	[1]	0x0: No error 0x1: Data array ECC/parity error on read - Writing 1 clears the error bit. - Writing 0 has no effect on the error bit.
BF	[2]	0x0: No error 0x1: External memory bus error on memory write - Writing 1 clears the error bit. - Writing 0 has no effect on the error bit.
Auxiliary Operation Errors		
SE	[3]	0x0: No error 0x1: Status array ECC/parity error on read - Writing 1 clears the error bit. - Writing 0 has no effect on the error bit.
TE	[4]	0x0: No error 0x1: Tag array ECC/parity error on read - Writing 1 clears the error bit. - Writing 0 has no effect on the error bit.
DE	[5]	0x0: No error 0x1: Data array ECC/parity error on read. - Writing 1 clears the error bit. - Writing 0 has no effect on the error bit.
RE	[6]	0x0: No error 0x1: ECC/parity error on restoring of aux registers from deep sleep mode. - Writing 1 clears the error bit. - Writing 0 has no effect on the error bit.



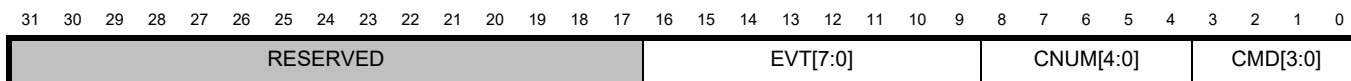
- Until the registered fault status bits in the `SLC_AUX_FAULT_STAT` register are cleared, only the oldest error is captured; subsequent errors are not captured.
- The interrupt signal is routed to the top level and is unconnected. This design provides the user the flexibility to connect the signal to a proper interrupt controller or master core.

19.32 L2 Cache Performance Monitor Command Register, SLC_AUX_PM_CMD

Address: 0x926

Access: RW

Figure 19-30 SLC_AUX_PM_CMD Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Use this register to enable or disable counters for specific events. The events are specified in [Table 19-12](#).

Table 19-12 SLC_AUX_PM_CMD Field Description

Field	Bit	Description
CMD	[3:0]	Performance monitor command 0x0 : Enable the specified counter, CNUM[4:0], to count the event EVT[7:0] 0x1: Disable the specified counter, CNUM[4:0] 0x2 : Read the specified counter and event, CNUM[4:0], into the SLC_AUX_PM_CNT0 and SLC_AUX_PM_EVENT registers 0x3: Read and clear the specified counter, CNUM[4:0] into the SLC_AUX_PM_CNT0 register. - Clear the corresponding overflow bit in the SLC_AUX_PM_OVF register. - Read the event is into the SLC_AUX_PM_EVENT register 0x4: The specified counter, CNUM[4:0], is preset with the count value in the SLC_AUX_PM_CNT0 register 0x5: Disable all counters 0x6 : Clear all counters and overflow register 0x7: Reserved.
CNUM	[8:4]	Counter number for the above command (CMD[3:0])
EVT	[16:9]	Event number to enable for the counter, up to 512 different events can be specified

**Note**

When you execute the `SLC_AUX_PM_CMD[3:0] == 0011` or `SLC_AUX_PM_CMD[3:0] == 0010` command, the `SLC_AUX_PM_CNT0`, `SLC_AUX_PM_EVENT`, and `SLC_AUX_PM_OVF` registers are not immediately updated. Therefore, if you issue these (`SLC_AUX_PM_CMD[3:0] == 0011` or `SLC_AUX_PM_CMD[3:0] == 0010`) commands and immediately read these performance monitor event counter or performance monitor counter registers, invalid values are returned. Best practice is to repeat the LR instruction on these registers (`SLC_AUX_PM_CNT0`, `SLC_AUX_PM_CNT1`, `SLC_AUX_PM_EVENT`, and `SLC_AUX_PM_OVF`).

For example:

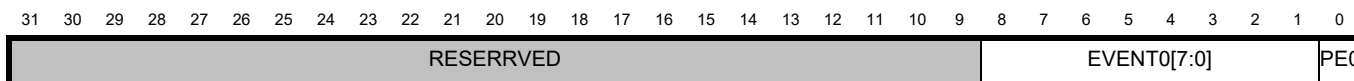
```
sr          r1, [00000926]
lr          r2, [00000927]
lr          r2, [00000927]
```

19.33 L2 Cache Performance Monitor Event Register, SLC_AUX_PM_EVENT

Address: 0x927

Access: R

Figure 19-31 SLC_AUX_PM_EVENT Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Use this register to read the status of a counter (enabled or disabled) and the event associated with the counter. The information in this register relates to the counter specified by the SLC_AUX_PM_CMD.CNUM[4:0] field. To read this register, do the following:

1. Write the SLC_AUX_PM_CMD register with suitable values for the following fields:

- CMD[3:0] with 4'b0010 or 4'b0011.
- CNUM[4:0] with intended counter number.

The event value is copied to the this register by the L2 cache.

2. Read the value from the SLC_AUX_PM_EVENT register.

Table 19-13 SLC_AUX_PM_EVENT Field Description

Field	Bit	Description
PE0	[0]	Returns the enable/disable status of a performance monitor event counter. ■ 0x0: performance monitor event counter is disabled (default) ■ 0x1: the performance monitor event counter is enabled for the event specified in the EVENT field
EVENT	[8:1]	The specific event that is being counted, up to 512 different events can be specified, see Table 19-14 .

Table 19-14 Performance Counter Event Listing

EVENT	Description
0000000	Total requests on port 0
0000001	Total requests on port 1
0000010	Total requests on port 2
0000011	Total requests on port 3

Table 19-14 Performance Counter Event Listing

EVENT	Description
0000100	Total requests on port 4
0000101	Total requests on port 5
0000110	Total requests on port 6
0000111	Total requests on port 7
0001000	Total requests on port 8
0001001	Total requests on port 9
0001010	Total requests on port 10
0001011	Total requests on port 11
0001100	Total requests on port 12
0001101	Total requests on port 13
0001110	Total requests on port 14
0001111	Total requests on port 15
0010000	Total core requests from all ports
0010001	Total read requests from all ports
0010010	Total write requests from all ports
0010011	Total tag miss
0010100	Read tag miss
0010101	Write tag miss
0010110	Total stall cycles due to tag queue full
0010111	Number of requests with tag queue full
0011000	Total stall cycles due to data queue full
0011001	Number of requests with data queue full
0011010	Total stall cycles due to status queue full
0011011	Number of requests with status queue full
0011100	Total stall cycles for input requests for all ports
0011101	Total stall cycles for tag miss due to data miss queue full
0011110	Total miss requests stall due to eviction buffer full
0011111	Total number of 32-byte evictions; multiple 32-byte per cache line

Table 19-14 Performance Counter Event Listing

EVENT	Description
0100000	Total number of cache line evictions
0100001	Total number of stalled cycles for input requests due to data buffer full
0100010	Total number of stalled requests due to data buffer full
0100011	Total stalled write requests due to second write request from the same port
0100100	Total stalled requests due to multiple requests with the same tag index
0100101	Total cache miss requests when all ways are locked; read requests only, write requests is error
0100110	Count of concurrent second requests to the same cache line as pending request
0100111	Count of concurrent second requests to the same cache line as pending tag-miss request
0101000	Count of concurrent second read requests to the same cache line as pending tag-miss request
0101001	Count of concurrent second write requests to the same cache line as pending tag-miss request
0101010	Count of data prefetch requests which already existed in L2 cache
0101011	Count of instruction prefetch requests which already existed in L2 cache
0101100	Reserved
0101101	Reserved
0101110	Reserved
0101111	Reserved
0110000	Total stall requests for port 0
0110001	Total stall requests for port 1
0110010	Total stall requests for port 2
0110011	Total stall requests for port 3
0110100	Total stall requests for port 4
0110101	Total stall requests for port 5

Table 19-14 Performance Counter Event Listing

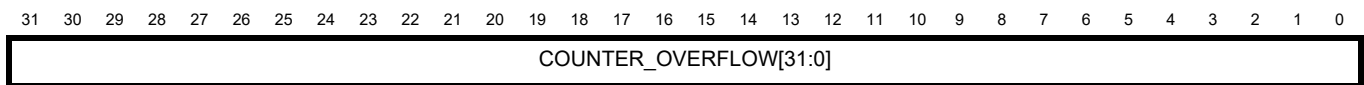
EVENT	Description
0110110	Total stall requests for port 6
0110111	Total stall requests for port 7
0111011	Total stall requests for port 8
0111001	Total stall requests for port 9
0111010	Total stall requests for port 10
0111011	Total stall requests for port 11
0111100	Total stall requests for port 12
0111101	Total stall requests for port 13
0111110	Total stall requests for port 14
0111111	Total stall requests for port 15

19.34 L2 Cache Performance Counter Overflow Register, SLC_AUX_PM_OVF

Address: 0x928

Access: RW

Figure 19-32 SLC_AUX_PM_OVF Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Overflow bit per performance counter:

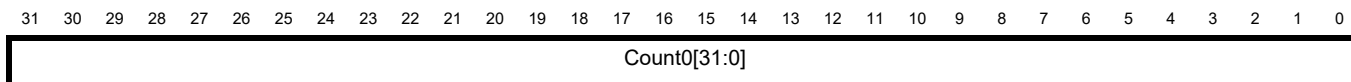
- A write of 1 clears the overflow bit.
- A write 0 has no effect on the overflow bit.

19.35 L2 Cache Performance Monitor Count0 Register, SLC_AUX_PM_CNT0

Address: 0x929

Access: RW

Figure 19-33 SLC_AUX_PM_CNT0 Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

Up to 32 pairs of counters can be implemented as specified by `SLC_AUX_CACHE_CONFIG.PM_NUM[4:0]` field. For 48-bit or 64-bit counters, the `SLC_AUX_PM_CNT1` register is also used.

The event counter registers can be used to preset the count value for a specific counter.

1. Write the value to be preset into the event counter register.
2. Write the `SLC_AUX_PM_CMD` register with suitable values for the following fields:
 - `CMD[3:0]` with 0x4
 - `CNUM[4:0]` with the intended counter number

Read the current count of a counter to the event count register by issuing the read command in the `SLC_AUX_PM_CMD` register.

1. Write the Performance Monitor Command register with suitable values for the following fields:
 - `CMD[3:0]` with 0x2 or 0x3.
 - `CNUM[4:0]` with the intended counter number.

The counter value is copied to the event counter register by the L2 cache.

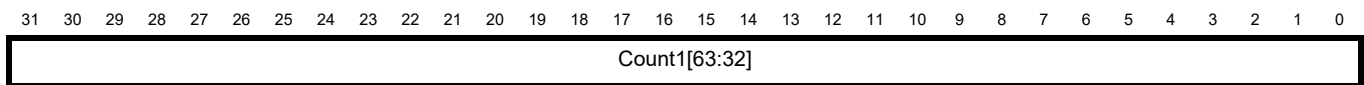
2. Read the value from the event counter register.

19.36 L2 Cache Performance Monitor Count1 Register, SLC_AUX_PM_CNT1

Address: 0x92A

Access: RW

Figure 19-34 SLC_AUX_PM_CNT1 Register



The L2 cache registers are physically located in the L2 cache, and are shared by all the cores.

This register is used only for a 48-bit counter or a 64-bit counter. The bits[15:0] are needed for a 48-bit counter; while all 32-bits are needed for 64-bit counter.

The event counter registers can be used to preset the count value for a specific counter.

1. Write the value to be preset into the event counter register.
2. Write the SLC_AUX_PM_CMD register with suitable values for the following fields:
 - CMD [3 : 0] with 0x4
 - CNUM [4 : 0] with the intended counter number

Read the current count of a counter to the event count register by issuing the read command in the SLC_AUX_PM_CMD register.

1. Write the Performance Monitor Command register with suitable values for the following fields:
 - CMD [3 : 0] with 0x2 or 0x3.
 - CNUM [4 : 0] with the intended counter number.

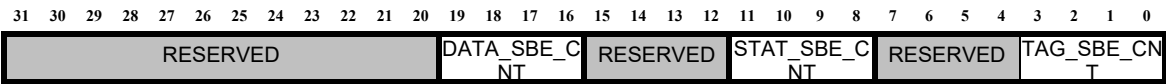
The counter value is copied to the event counter register by L2 cache.

2. Read the value from the event counter register.

19.37 L2 Cache SBE Counter Register, SLC_AUX_ECC_SBE_CNT

Address:	0x93C
Additional Information:	This register is implemented in the cluster auxiliary space. This register is shared by all the cores.
Access:	RW
Reset	0x0000_0000

Figure 19-35 SLC_AUX_ECC_SBE_CNT Register



This register records the number of single-bit errors that have been corrected during L2 cache tag, data, and status reads.



Note

This register exists in the processor only when ECC in SLC is configured for SECCDED. This register does not exist in the processor when either ECC is not configured or only parity protection is configured.

Table 19-15 SLC_AUX_ECC_SBE_CNT Bit Field Description

Field	Bit	Description
TAG_SBE_CNT	[3:0]	ECC single-bit correction error counter during L2 cache tag reads. - Software must write a 0 to clear this register. Values other than 0 are ignored. - If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. - When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.

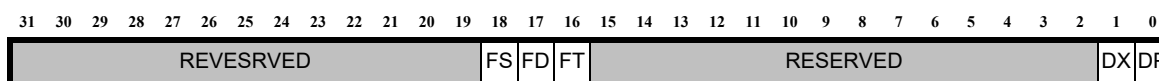
Table 19-15 SLC_AUX_ECC_SBE_CNT Bit Field Description

STAT_SBE_CNT	[11:8]	ECC single-bit correction error counter during L2 cache status reads. ■ Software must write a 0 to clear this register. Values other than 0 are ignored. ■ When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.
DATA_SBE_CNT	[19:16]	ECC single-bit correction error counter during L2 cache data reads. ■ Software must write a 0 to clear this register. Values other than 0 are ignored. ■ When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.

19.38 L2 Cache ECC Control Register, SLC_AUX_ECC_CTRL

Address:	0x93D
Additional Information:	This register is implemented in the cluster auxiliary space. This register is shared by all the cores.
Access:	RW
Reset	0x0000_0000

Figure 19-36 SLC_AUX_ECC_CTRL Register



Use this register to enable or disable ECC and parity protection on the L2 cache. This register also includes the error status bits that are updated when a double-bit error or parity error occurs in the L2 cache TAG, DATA, and STATUS RAMs. This register is present only when error protection is configured in the processor.

Table 19-16 SLC_AUX_ECC_CTRL Bit Field Description

Field	Bit	Description
DP	[0]	Control bit for ECC and parity protection for the L2 cache. 0x0: Enables protection 0x1: Disables protection; if protection is disabled: - the processor does not raise exceptions regardless of the DX bit's value. - ECC or parity logic is disabled and errors are neither detected nor corrected. The FT, FD, and FS flags are not set. - Error reporting on <code>slc_intr_valid</code> is disabled. On reset, ECC and parity protection is enabled.

Table 19-16 SLC_AUX_ECC_CTRL Bit Field Description

DX	[1]	Control bit for generation or suppression of exceptions for the L2 cache. ■ 0x0: Enables exceptions. The processor raises exceptions when multi-bit errors or parity errors are detected. ■ 0x1: Disables exceptions; if error protection is enabled and exceptions are disabled: - ECC logic detects and corrects single-bit errors. Memory error exceptions are not raised when a multi-bit error or parity error is detected. - Error reporting on <code>slc_intr_valid</code> is disabled. - Imprecise errors are detected but are not reported on <code>slc_intr_valid</code> and <code>SLC_AUX_FAULT_STAT</code> is not updated On reset, exceptions are enabled.
FT	[16]	Indicates if ECC or parity errors have occurred in the TAG RAM ■ 0x0: No error is detected ■ 0x1: Double-bit error or parity error is detected in the TAG RAM. The FT bit can be set only by the ECC hardware logic; the software cannot write 1 to this bit. The software must write 0 using the SR instruction to clear this bit.
FD	[17]	Indicates if ECC or parity errors have occurred in the DATA RAM ■ 0x0: No error is detected ■ 0x1: Double-bit error or parity error is detected in the DATA RAM. The FD bit can be set only by the ECC hardware logic; the software cannot write 1 to this bit. The software must write 0 using the SR instruction to clear this bit.
FS	[18]	Indicates if ECC or parity errors have occurred in the STATUS RAM ■ 0x0: No error is detected ■ 0x1: Double-bit error or parity error is detected in the STATUS RAM. The FS bit can be set only by the ECC hardware logic; the software cannot write 1 to this bit. The software must write 0 using the SR instruction to clear this bit.

19.39 L2 Cache Build Configuration Register, SLC_BUILD

Address: 0xCE

Access: R

Figure 19-37 SLC_BUILD Register



This register is included in a core whenever the core is configured in a cluster where an L2 cache is present.

Table 19-17 L2_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number 0x1: Initial version of the L2 cache 0x2: Second version of the L2 cache 0x3: Includes support for sharing data memory with CSM, support for 64-bit ECC (data memory only), support for single-bit error correction ECC counters 0x4: Indicates the presence of the SLC_AUX_ECC_CTRL register and support for all zero and one detection.
NUMBER	[15:8]	Identity of L2 cache Currently this field value is fixed to 0x1.

19.40 Exceptions

See [Table 5-4](#) on page [233](#).

20

Shared Memory Configuration

20.1 Introduction

When the shared memory configuration is enabled in the processor, the L2 cache RAMs are implemented as a shared memory divided into four quadrants, and each quadrant can be selected as either Level-2 cache or cluster shared memory (CSM). The L2 cache is selected from first quadrant to fourth quadrant in the shared memory while CSM is configured from the fourth quadrant to first quadrant. This shared memory configuration has the following restrictions:

- The shared-memory size must be the same size as the L2-cache memory size.
- If ECC is configured, the L2 cache and the CSM must use 64-bit ECC.
- When the shared memory configuration is used in your processor, the L2 cache line size is fixed to 64 bytes, the number of L2 cache banks is fixed to 8, and the minimum data-latency cycle is 3.
- You can use a shared memory only if you include both L2 cache and CSM in your processor configuration.
- The CSM memory size can either be 1/4th, 1/2, or full of the shared memory size.
- If a quadrant is not used by either L2 cache or CSM, by default it is assigned to the L2 cache.
- The L2 cache or CSM has the option of powering down the allocated quadrants of shared memory when not in use.

20.2 Configuring the Shared Memory

In ARChitect, specify the following option to configure shared memory in the processor:

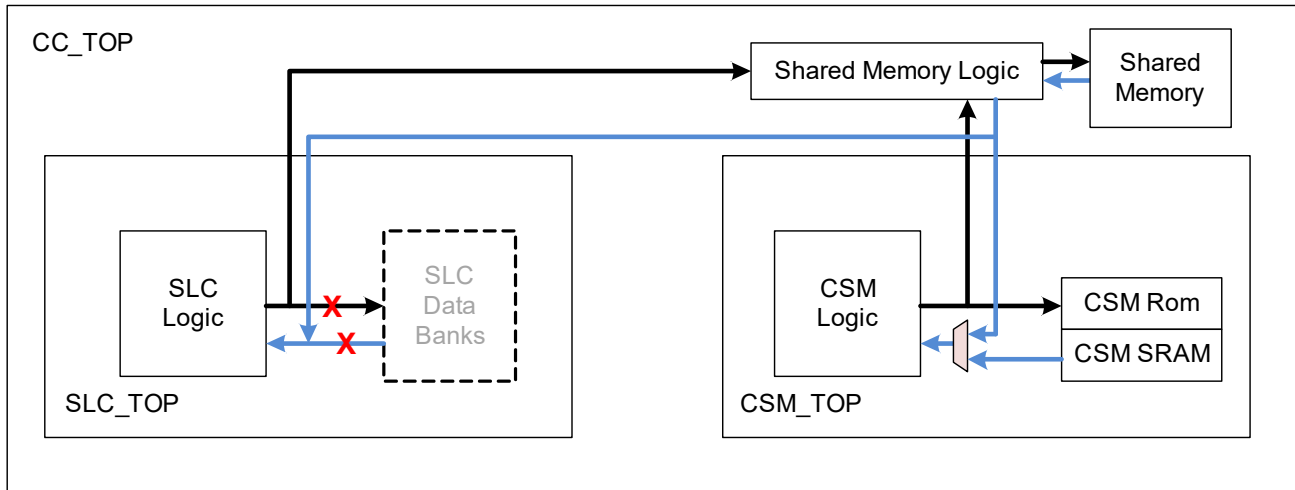
```
-cc_allow_csm_l2cache_share_memory
```

20.3 Programming the Shared Memory

You can program shared memory using the [SM_CONFIG_REG](#) register.

20.4 Programming Considerations

- If half of the shared memory (S2 and S3) is allocated to CSM, that half must be locked by L2 cache. It is responsibility of the software to never unlock the S2 and S3 in SLC. From the L2 cache perspective, shared memory configured as CSM is not accessible for anything including power management.



- You must program the shared memory configuration register immediately after system power up as part of the boot code or part of the initialized sequence of the system. Programming this register at any other time (other than at startup) causes unexpected behavior.
 - The boot code must be placed in the CSM or a private memory so that fetching the boot code and modifying the configuration of shared memory do not result in sending requests to the shared memory. Also, the way_off bits of the L2 cache control register must be modified according to the shared memory configuration for L2 cache. The code to modify the L2 cache control register must also be in the CSM or the private memory.
- L2 cache is configured only with two options: S3:S0 = xx00 and 0000.
- CSM is configured with 3 options: S3:S0 = 1xxx and 11xx, 1111.
- This register must be in sync with the L2 cache and CSM programming. Writing and reading of the unallocated memory by the L2 cache or CSM results undefined behavior. One of the many undefined behaviors is the assertion of read error when ECC is enabled.

Table 20-1 Shared Memory Configurations

S0	S1	S2	S3	Comments
L2 cache	L2 cache	L2 cache	L2 cache	All four quadrants are used by the L2 cache.
L2 cache	L2 cache	L2 cache (powered off)	CSM	■ Quadrants S0 and S1 are used by the L2 cache. ■ Quadrant S2 is not used and is forbidden to access. ■ Quadrant S3 used by CSM.
L2 cache	L2 cache	CSM	CSM	■ Quadrants S0 and S1 are used by the L2 cache. ■ Quadrants S3 and S2 are used by CSM.

Table 20-1 Shared Memory Configurations

S0	S1	S2	S3	Comments
L2 cache (powered off)	L2 cache (powered off)	L2 cache (powered off)	CSM	■ Quadrant S3 is used by CSM. ■ Access to the S2 quadrant is forbidden and usually powered-off. ■ Quadrants S1 and S0 are used as L2 cache, and can be powered-off if not used. ■ The <code>SLC_AUX_CACHE_CTRL.CD</code> must be set to 0x1. ■ The BIU routes all the L2 cache addresses directly to the external memory.
L2 cache (powered off)	L2 cache (powered off)	CSM	CSM	■ Quadrants S3 and S2 are used by CSM. ■ Quadrants S1 and S0 are used as L2 cache can be powered-off if not used. ■ The <code>SLC_AUX_CACHE_CTRL.CD</code> must be set to 0x1. ■ The BIU routes all the L2 cache addresses directly to the external memory.
CSM	CSM	CSM	CSM	■ All four quadrants (S3, S2, S1, S0) used by CSM. ■ The <code>SLC_AUX_CACHE_CTRL.CD</code> must be set to 0x1. ■ The BIU routes all the L2 cache addresses directly to the external memory.

20.5 Shared Memory Configuration Register, SM_CONFIG_REG

Address: 0x9A8

Access: RW

Figure 20-1 SM_CONFIG_REG

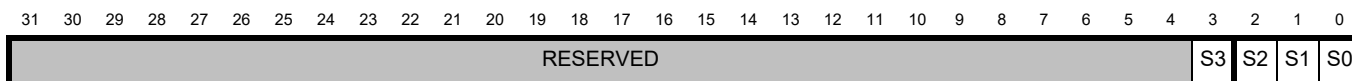


Table 20-2 Shared Memory Configuration Register Field Description

Field	Bit	Description
S0	[0]	Specify the memory allocation for the first quadrant of the shared memory: 0x0 (default): allocate this quadrant to the L2 cache. 0x1: allocate this quadrant to CSM. The corresponding <code>way_off[0]</code> of the L2 cache control register (0x903) must be set to 1.
S1	[1]	Specify the memory allocation for the first quadrant of the shared memory: 0x0 (default): allocate this quadrant to the L2 cache. 0x1: allocate this quadrant to CSM. The corresponding <code>way_off[1]</code> of the L2 cache control register (0x903) must be set to 1.
S2	[2]	Specify the memory allocation for the first quadrant of the shared memory: 0x0 (default): allocate this quadrant to the L2 cache. 0x1: allocate this quadrant to CSM. The corresponding <code>way_off[2]</code> of the L2 cache control register (0x903) must be set to 1.
S3	[3]	Specify the memory allocation for the first quadrant of the shared memory: 0x0 (default): allocate this quadrant to the L2 cache. 0x1: allocate this quadrant to CSM. The corresponding <code>way_off[3]</code> of the L2 cache control register (0x903) must be set to 1.

21

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

21.1 Auxiliary Registers

Table 21-1 Error Protection Auxiliary Registers

Address	Auxiliary Register Name	Description
0x3C	ECC_SBE_COUNT	Single-bit error counter
0x3F	ERP_CTRL	Error protection control register
0xC7	ERP_BUILD	Build Configuration Register: error protection

21.1.1 ECC SBE Counter Register, ECC_SBE_COUNT

Address: 0x3C

Access: RW

Reset 0x0000_0000

Figure 21-1 ECC_SBE_COUNT Register

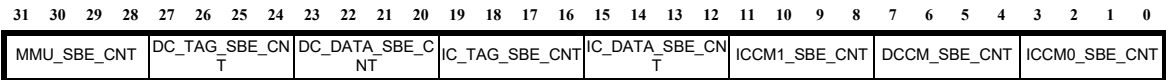
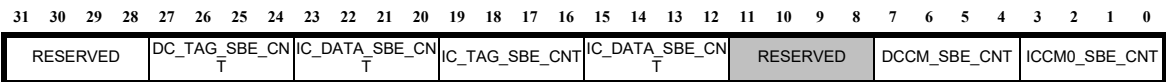


Figure 21-2 ECC_SBE_COUNT Register



This register records the number of single-bit errors that have been corrected in the memories. The counter for the CSM memories is in the “CSM_SBE_CNT” register, and the L2 cache counter is the SLC_AUX_ECC_SBE_CNT register. Certain bit fields exist only when the related component is present in the build.

Table 21-2 ECC_SBE_COUNT Bit Field Description

Field	Bit	Reset	Description
ICCM0_SBE_CNT	[3:0]	0	ECC single-bit correction error counter for ICCM0. ■ Software must write a 0 to clear this register. Values other than 0 are ignored. ■ If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. ■ When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.

Table 21-2 ECC_SBE_COUNT Bit Field Description

DCCM_SBE_CNT	[7:4]	0	ECC single-bit correction error counter for DCCM. ■ Software must write a 0 to clear this register. Values other than 0 are ignored. ■ If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. ■ When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.
ICCM1_SBE_CNT	[11:8]	0	ECC single-bit correction error counter for ICCM1. ■ Software must write a 0 to clear this register. Values other than 0 are ignored. ■ If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. ■ When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.
IC_DATA_SBE_CNT	[15:12]	0	ECC single-bit correction error counter for the instruction cache data memory. ■ Software must write a 0 to clear this register. Values other than 0 are ignored. ■ If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. ■ When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.

Table 21-2 ECC_SBE_COUNT Bit Field Description

IC_TAG_SBE_CNT	[19:16]	0	ECC single-bit correction error counter for the instruction cache tag. - Software must write a 0 to clear this register. Values other than 0 are ignored. - If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. - When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.
DC_DATA_SBE_CNT	[23:20]	0	ECC single-bit correction error counter for the data cache data memory. - Software must write a 0 to clear this register. Values other than 0 are ignored. - If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. - When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again. This field is available only when single-bit error correction and double-bit error detection is enabled on this memory.
DC_TAG_SBE_CNT	[27:24]	0	ECC single-bit correction error counter for the data cache tag memory. - Software must write a 0 to clear this register. Values other than 0 are ignored. - If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. - When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again.
MMU_SBE_CNT	[31:28]	0	ECC single-bit correction error counter for the MMU. - Software must write a 0 to clear this register. Values other than 0 are ignored. - If data ECC protection is not configured, this register always returns 0, and writes to this field are ignored. - When this counter reaches the maximum value (0xF), no new single-bit error corrections are counted. This counter must be cleared before it can start recording the single-bit corrections again.
RESERVED	[31:28]	0	Reserved. Read-as-zero; ignored on write.

21.1.2 Error Protection Hardware Control Register, ERP_CTRL

Address: 0x3F

Access: RW

Reset 0x00000000

Figure 21-3 ERP_CTRL Register

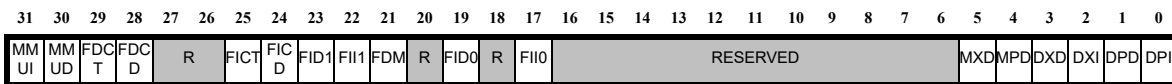
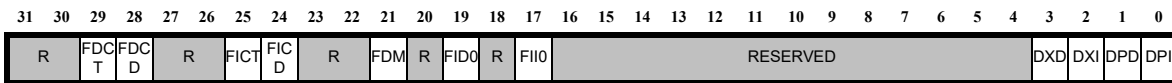


Figure 21-4 ERP_CTRL Register



Note

Bits marked as RESERVED or R are read as zero and ignored on write.

This register allows you to enable or disable ECC and parity protection on instruction, DMP, and MMU paths. This register is updated when a double-bit error occurs in the instruction path or the DMP path.

Instruction path includes ICCM0, ICCM1, instruction cache data, and instruction cache tag data. DMP path includes DCCM, Data cache data, and data cache tag data. This register is only present when `ECC_OPTION=1|2`. Certain bit fields exist only when the related component is present in the build.

When the configuration option `-pipe_edc_option==true`, any single or double-bit errors in the control bits of this register (`ERP_CTRL[5:0]`) are detected, and the signal `edc_pipe_err` is asserted high.

Table 21-3 ERP_CTRL Bit Field Description

Field	Bit	Reset	Description
DISABLE_PROT_IFU (DPI)	[0]	0	Control bit for ECC/parity protection on instruction path; A value of 0 enables protection, and 1 disables protection. On reset, the control bits are set to 0; that is ECC/parity protection is enabled.
DISABLE_PROT_DMP (DPD)	[1]	0	Control bit for ECC/parity protection on DMP; A value of 0 enables protection, and 1 disables protection. When ECC is disabled, the read-modify-write operations are not supported. On reset, the control bits are set to 0; that is ECC/parity protection is enabled.

Table 21-3 ERP_CTRL Bit Field Description

Field	Bit	Reset	Description
DISABLE_EXCP_IFU (DXI)	[2]	0	Control bit for enabling/disabling protection exceptions on instruction path; A value of 0 enables exceptions, and 1 disables exceptions for two-bit data and address errors. On reset, the control bits are set to 0; that is exceptions are enabled.
DISABLE_EXCP_DMP (DXD)	[3]	0	Control bit for enabling/disabling protection exceptions on DMP; A value of 0 enables exceptions, and 1 disables exceptions for two-bit data and address errors. On reset, the control bits are set to 0; that is exceptions are enabled.
ECC_MMU_DISABLE (MPD)	[4]	0	Control bit for ECC/parity protection on MMU; A value of 0 enables protection, and 1 disables protection. On reset, the control bits are set to 0; that is ECC/parity protection is enabled
ECC_MMU_EXPN_DISABLE(MXD)	[5]	0	Control bit for enabling/disabling protection exceptions on MMU; A value of 0 enables exceptions, and 1 disables exceptions for two-bit data and address errors. On reset, the control bits are set to 0; that is exceptions are enabled.
FLT_ECC_ICCM0_INST_DATA (FII0)	[17]	0	Set to 1 indicates double-bit error on ICCM0 instruction word; Write 0 to clear this bit.
FLT_ECC_ICCM0_DATA (FID0)	[19]	0	Set to 1 indicates error on ICCM0 DMP data word; write 0 to clear this bit.
FLT_ECC_DCCM_DATA (FDM)	[21]	0	Set to 1 indicates double-bit error on DCCM data word; write 0 to clear this bit.
FLT_ECC_ICCM1_INST_DATA (FII1)	[22]	0	Set to 1 indicates double-bit error on ICCM1 instruction word; write 0 to clear this bit.
FLT_ECC_ICCM1_DATA (FID1)	[23]	0	Set to 1 indicates error on ICCM1 DMP data word; write 0 to clear this bit.
FLT_ECC_ICACHE_INST_DATA (FICD)	[24]	0	Set to 1 indicates: double-bit error on ICACHE instruction word; write 0 to clear this bit.
FLT_ECC_ICACHE_TAG_DATA (FICT)	[25]	0	Set to 1 indicates double-bit error on ICACHE_TAG; write 0 to clear this bit.
FLT_ECC_DCACHE_INST_DATA (FDCT)	[28]	0	Set to 1 indicates double-bit error on data word in DCACHE data word; write 0 to clear this bit.
FLT_ECC_DCACHE_TAG_DATA (FDCT)	[29]	0	Set to 1 indicates double-bit error on data in DCACHE_TAG word; write 0 to clear this bit.
FLT_ECC_MMU_NTLB_DTLB (MMUD)	[30]	0	Set to 1 indicates double-bit error accessing NTLB during DTLB update; write 0 to clear this bit.

Table 21-3 ERP_CTRL Bit Field Description

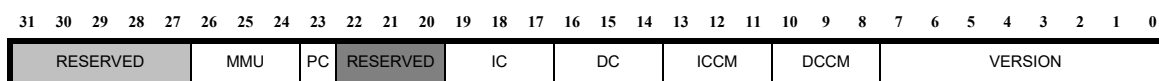
Field	Bit	Reset	Description
FLT_ECC_MMU_NTLB_ITLB (MMUI)	[31]	0	Set to 1 indicates double-bit error during MMU auxiliary operations or accessing NTLB during ITLB update; write 0 to clear this bit.

21.1.3 Error Protection Configuration Register, ERP_BUILD

Address: 0xC7

Access: R

Figure 21-5 ERP_BUILD Register



The ERP_BUILD register contains information about the ECC and parity 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.

[Table 21-4](#) lists the field descriptions:

Table 21-4 ERP_BUILD Register

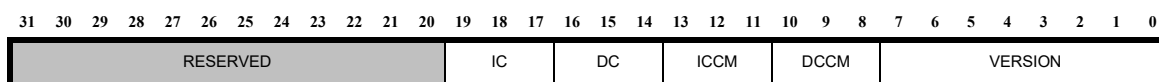
Field	Bit	Description
VERSION	[7:0]	Indicates the ECC Version. The current value is 5. Includes support for the following: ■ Added option to independently configure Error Protection for memories ■ Added ECC Single bit error counters ■ Support for error detection on Program Counter, Register File (32 general-purpose core registers) ■ Added option to detect all zeros and all ones. ■ <code>ecc_syndrome_option</code> feature is added to expose syndrome bits in case of single and double bit errors
DCCM	[10:8]	Indicates ECC or parity protection on DCCM. ■ 0x0: indicates no protection is enabled ■ 0x1: indicates ECC protection on data is enabled ■ 0x2: indicates ECC protection on data and address is enabled ■ 0x3: Parity protection on Data ■ 0x4: Parity Protection on Address and Data ■ 0x5: ECC protection on Data with all zero/one detection ■ 0x6: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved.

Table 21-4 ERP_BUILD Register

Field	Bit	Description
ICCM	[13:11]	Indicates ECC or parity protection on ICCMs. ■ 0x0: indicates no protection is enabled ■ 0x1: indicates ECC protection on data is enabled ■ 0x2: indicates ECC protection on data and address is enabled ■ 0x3: Parity protection on Data ■ 0x4: Parity Protection on Address and Data ■ 0x5: ECC protection on Data with all zero/one detection ■ 0x6: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved. Note: ICCM0 and ICCM1 must have the same error protection configuration.
DC	[16:14]	Indicates ECC or parity protection on data cache. ■ 0x0: indicates no protection is enabled ■ 0x1: indicates ECC protection on data is enabled ■ 0x2: indicates ECC protection on data and address is enabled ■ 0x3: Parity protection on Data ■ 0x4: Parity Protection on Address and Data ■ 0x5: ECC protection on Data with all zero/one detection ■ 0x6: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved.
IC	[19:17]	Indicates ECC or parity protection on instruction cache. ■ 0x0: indicates no protection is enabled ■ 0x1: indicates ECC protection on data is enabled ■ 0x2: indicates ECC protection on data and address is enabled ■ 0x3: Parity protection on Data ■ 0x4: Parity Protection on Address and Data ■ 0x5: ECC protection on Data with all zero/one detection ■ 0x6: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved.

Table 21-4 ERP_BUILD Register

Field	Bit	Description
PC	[23]	Indicates EDC (error detection) protection on Program Counter, Register File (32 general-purpose core registers), and ERP_CTRL[5:0] register. 0x0: indicates EDC protection is not configured 0x1: indicates EDC protection is configured When EDC protection is configured, single and double-bit errors are detected and an external signal <code>edc_pipe_err</code> is asserted high. Exceptions are not generated for these errors. This bit reflects the value of the configuration option <code>-pipe_edc_option</code>.
MMU	[26:24]	Indicates ECC on MMU. 0x0: indicates no protection is enabled 0x1: indicates ECC protection on data is enabled 0x2: indicates ECC protection on data and address is enabled 0x3: Parity protection on Data 0x4: Parity Protection on Address and Data 0x5: ECC protection on Data with all zero/one detection 0x6: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved.

Figure 21-6 ERP_BUILD Register

The ERP_BUILD register contains information about the ECC and parity 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.

[Table 21-4](#) lists the field descriptions:

Table 21-5 ERP_BUILD Register

Field	Bit	Description
VERSION	[7:0]	Indicates the ECC Version. All non-enumerated values are reserved. ■ 0x7: VEC_ECC_CTRL (0x549) aux, now has a disable exception (DE) field that allows the exceptions to be disabled and ECC_VEC_MEM_SBE_CNT field to indicate the count of single bit errors ■ 0x5: includes support for the following: - Added option to independently configure Error Protection for memories - Added ECC Single bit error counters - Added option to detect all zeros and all ones. - <code>ecc_syndrome_option</code> feature is added to expose syndrome bits in case of single and double bit errors
DCCM	[10:8]	Indicates ECC or parity protection on DCCM. ■ 000: indicates no protection is enabled ■ 001: indicates ECC protection on data is enabled ■ 010: indicates ECC protection on data and address is enabled ■ 011: Parity protection on Data ■ 100: Parity Protection on Address and Data ■ 101: ECC protection on Data with all zero/one detection ■ 110: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved.
ICCM	[13:11]	Indicates ECC or parity protection on ICCMs. ■ 000: no protection is enabled ■ 001: ECC protection on data is enabled ■ 010: ECC protection on Address and Data ■ 011: Parity protection on Data ■ 100: Parity Protection on Address and Data ■ 101: ECC protection on Data with all zero/one detection ■ 110: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved.

Table 21-5 ERP_BUILD Register

Field	Bit	Description
DC	[16:14]	Indicates ECC or parity protection on data cache. ■ 000: no protection is enabled ■ 001: ECC protection on data is enabled ■ 010: ECC protection on Address and Data ■ 011: Parity protection on Data ■ 100: Parity Protection on Address and Data ■ 101: ECC protection on Data with all zero/one detection ■ 110: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved.
IC	[19:17]	Indicates ECC or parity protection on instruction cache. ■ 000: no protection is enabled ■ 001: ECC protection on data is enabled ■ 010: ECC protection on Address and Data ■ 011: Parity protection on Data ■ 100: Parity Protection on Address and Data ■ 101: ECC protection on Data with all zero/one detection ■ 110: ECC protection on Address and Data with all zero/one detection Other values for this field are reserved.

21.2 Exceptions

See [Table 5-4](#) on page [233](#).

22

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. You can configure up to two peripherals regions. Peripheral bus width of 32 bits or 64 bits are supported. When you configure two peripheral interfaces, you can configure the bus protocol and the bus width of the two peripheral buses independent of each other. The peripheral data bus can be configured independently from the memory bus protocol. The peripheral region can be changed at runtime through the DMP_PER_AUX register.

22.1 DMP Auxiliary Registers

Table 22-1 DMP Registers

Address	Auxiliary Register Name	Description
0x20A	DMP_PER_AUX	Peripheral memory region programming register
0xC3	DMP_PP_BUILD	Peripheral region build configuration register

22.1.1 Peripheral Memory Region, DMP_PER_AUX

Address: 0x20A

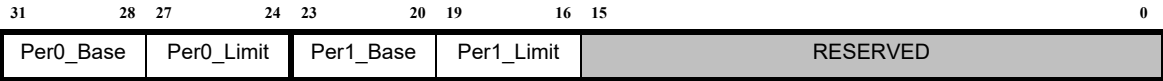
Access: RW

Default Build-time configurable

Figure 22-1 DMP_PER_AUX, start of peripheral region



Figure 22-2 DMP_PER_AUX, start of peripheral region



When at least one peripheral bus is enabled, the DMP_PER_AUX register exists, and identifies the mapping of all the peripheral buses to the memory regions. Memory regions mapped to peripheral buses are not cacheable.

 **Note**

To ensure a consistent view of data, the data cache must be flushed whenever DMP_PER_AUX is modified. If data caches are not configured, no flush operations are required.

In a multi-core processor implementations, this register exists in the cluster auxiliary space.

In a single-core processor configuration, this register always exists in the core auxiliary space.

-  **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

System Memory Map Considerations

In configurations with MMU, the peripheral region (which is associated with the external Peripheral Bus) can reside in the translated space (lower 2GB). In this case, for load and store (data) access, the address presented on the peripheral is the translated physical address (and may be 40 bits when PAE is configured).

However, the peripheral region is not visible to the Instruction Fetch Unit (similar to the DCCM, also invisible to IFU); and, IFU access to addresses (VAs) within the peripheral region, may be translated and target the system memory using the memory bus.

Thus, the peripheral region (decoded in VA) can appear differently for data versus instruction access. As the software tools generally assume a unified code and data memory space, it is suggested that the you avoid placing executed code in the peripheral region as it should be dedicated to data; this can be aided by ensuring that no TLB entries give execute permission for the Peripheral (VA) Region.

Programming Notes:

- Programming the limit to zero effectively sets the region from base to the highest memory address.
- The base address of the region is inclusive, that is, the region includes the base address.
- The limit is exclusive, that is, the region excludes this address.
- Avoid programming base to a value greater or equal than the limit. If so programmed, the region spans from the base to the highest memory address.

Figure 22-3 Peripheral Regions Based on Base and Limit Values

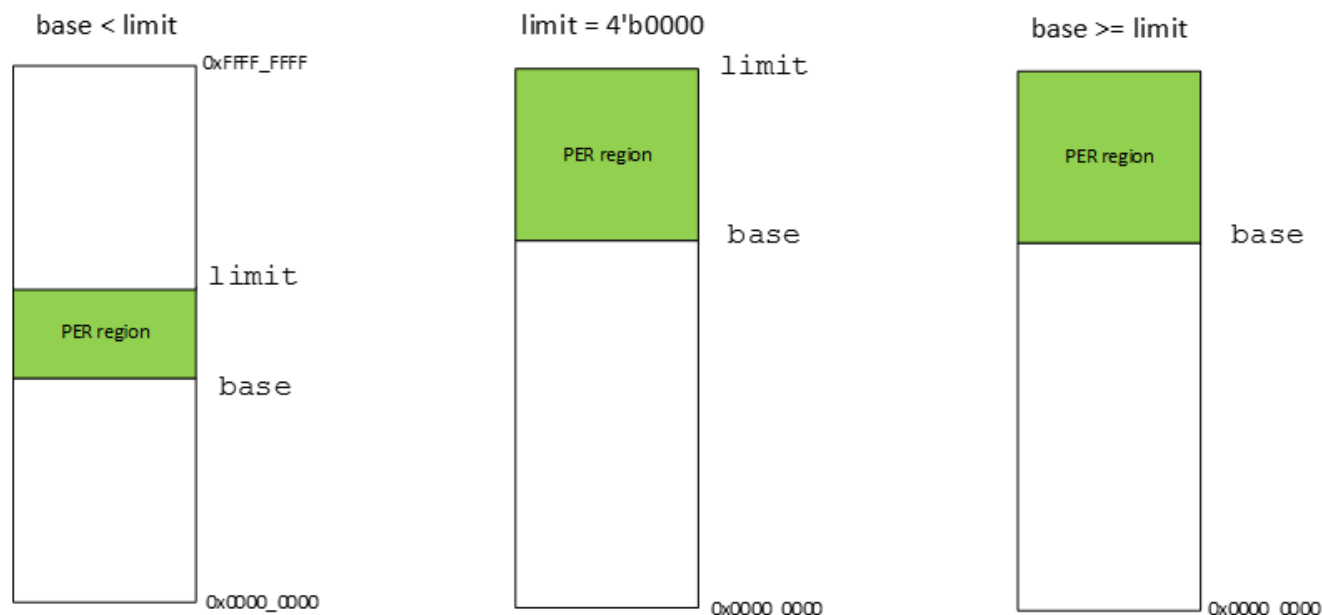


Table 22-2 DMP_PER_AUX Field Descriptions

Field	Bit	Description
RESERVED	[23:0][15:0]	Reserved. Read-as-zero; ignored on write.
per0_limit	[27:24]	Specifies the end of the peripheral aperture.
per0_base	[31:28]	Specifies the memory region of the peripheral aperture.
per1_limit	[27:24]	Specifies the end of the peripheral aperture. This options defaults to the configured value of the <code>-dmp_per1_limit</code> option.
per1_base	[31:28]	Specifies the memory region of the peripheral aperture. This options defaults to the configured value of the <code>-dmp_per1_base</code> option.

22.1.2 Peripheral Region Build Configuration Register, DMP_PP_BUILD

Address: 0xC3

Access: R

Figure 22-4 DMP_PP_BUILD Register

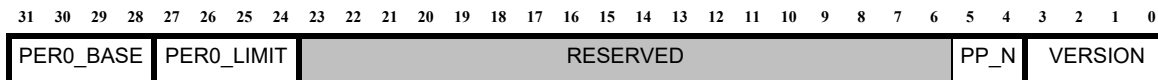
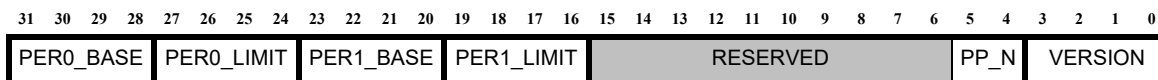


Figure 22-5 DMP_PP_BUILD Register



This register indicates the presence of a peripheral region, and its build-time configuration.

Table 22-3 DMP_PP_BUILD Field Descriptions

Field	Bit	Description
VERSION	[3:0]	Version of the peripheral region component. All non-enumerated values are reserved. 0x2: current version
PP_N	[5:4]	Number of peripheral buses. All non-enumerated values are reserved. 0x1: One peripheral region. 0x2: Two peripheral regions.
RESERVED	[23:6][15:6]	Reserved. Read-as-zero; ignored on write.
PER1_BASE	[19:16]	Specifies the memory region of the peripheral aperture. This options defaults to the configured value of the <code>-dmp_per1_base</code> option.
PER1_LIMIT	[23:20]	Specifies the end of the peripheral aperture. This options defaults to the configured value of the <code>-dmp_per1_limit</code> option.
PER0_LIMIT	[27:24]	Specifies the end of the peripheral aperture. This options defaults to the configured value of the <code>-dmp_per0_limit</code> option.
PER0_BASE	[31:28]	Specifies the memory region of the peripheral aperture. This options defaults to the configured value of the <code>-dmp_per0_base</code> option.

23

Write Buffer

23.1 Introduction

The write buffer is an optional component that buffers uncached stores and provides an early write response to the core. A total of three write buffers may exist in an ARC HS core, one for each interface (memory, peripheral 0, and peripheral 1).

23.2 Configuring the Write Buffer

For information about configuring the write buffer, see the *DesignWare ARC HS Series Databook*.

23.3 Write Buffer Auxiliary Registers

Table 23-1 Write Buffer Auxiliary Registers

Address	Auxiliary Register Name	Description
0x20C	Write Buffer Control Register, WB_CTRL	Write-buffer control register
0x20D	Write Buffer Status Register, WB_STATUS	Write-buffer status register
0xFD	Write Buffer BCR Register, WB_BUILD	Write-buffer build configuration register

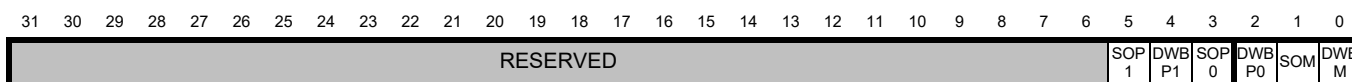
23.3.1 Write Buffer Control Register, WB_CTRL

Address: 0x20C

Access: RW

Reset {strict_ordering, disable} = {1, 0}. Configuration-dependent

Figure 23-1 WB_CTRL



This register allows you to control the enable/disable the write buffers, and the ordering of the read transactions on each write buffer.

Table 23-2 WB_CTRL Register Field Description

Field	Bit	Description
DWBM	[0]	Disable Write Buffer for Memory 0x1: disables the memory write buffer 0x0: enables the memory write buffer
SOM	[1]	Strict Ordering on Memory 0x1: read transactions are sent on the bus after all previous writes are completed 0x0: read transactions may be sent on the bus while previous writes are in progress. This mode relies on the memory subsystem never re-ordering reads with respect to writes
DWBP0	[2]	Disable Write Buffer for Peripheral 0 0x1: disables the peripheral 0 write buffer 0x0: enables the peripheral 0 write buffer
SOP0	[3]	Strict Ordering on Peripheral 0 0x1: read transactions are sent on the bus after all previous writes are completed 0x0: read transactions may be sent on the bus while previous writes are in progress. This mode relies on the memory subsystem never re-ordering reads with respect to writes
DWBP1	[4]	Disable Write Buffer for Peripheral 1 0x1: disables the peripheral 1 write buffer 0x0: enables the peripheral 1 write buffer

Table 23-2 WB_CTRL Register Field Description

Field	Bit	Description
SOP1	[5]	Strict Ordering on Peripheral 1 ■ 0x1: read transactions are sent on the bus after all previous writes are completed ■ 0x0: read transactions may be sent on the bus while previous writes are in progress. This mode relies on the memory subsystem never re-ordering reads with respect to writes

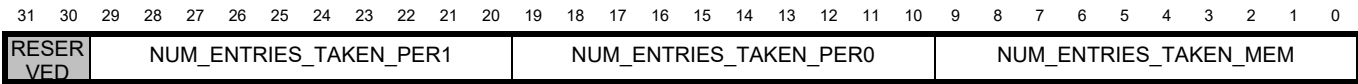
23.3.2 Write Buffer Status Register, WB_STATUS

Address: 0x20D

Access: R

Reset 0x0

Figure 23-2 WB_STATUS



This register records the number of write transactions that are pending in the write buffer.

Table 23-3 WB_STATUS Register Field Description

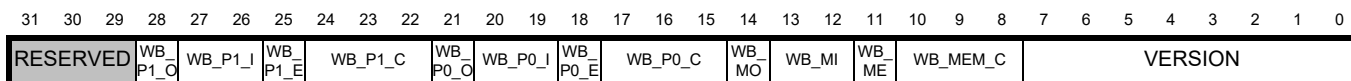
Field	Bit	Description
NUM_ENTRIES_TAKEN_MEM	[9:0]	Number of memory write transactions pending in the write buffer.
NUM_ENTRIES_TAKEN_PER0	[19:10]	Number of peripheral 0 write transactions pending in the write buffer.
NUM_ENTRIES_TAKEN_PER1	[29:20]	Number of peripheral 1 write transactions pending in the write buffer.

23.3.3 Write Buffer BCR Register, WB_BUILD

Address: 0xFD

Access: R

Figure 23-3 WB_BUILD



This register specifies the write buffer configuration.

Table 23-4 WB_BUILD Register Field Description

Field	Bit	Description
VERSION	[7:0]	0x1: initial version
WB_MEM_C	[10:8]	Write buffer memory capacity 0x0: no write buffer present 0x1: 16 entries 0x2: 32 entries 0x3: 64 entries 0x4: 128 entries 0x5: 256 entries 0x6: 512 entries 0x7: reserved
WB_ME	[11]	Write buffer memory write errors 0x0: write errors on this interface are not handled by the core. 0x1: write errors on this interface are handled by the core.
WB_MI	[13:12]	Write buffer memory implementation 0x0: flop-based implementation 0x1: one-cycle SRAM implementation 0x2: two-cycle SRAM implementation 0x3: reserved
WB_MO	[14]	Write buffer memory outstanding 0x0: 8 write transactions are outstanding on the bus. 0x1: 16 write transactions are outstanding on the bus.

Table 23-4 WB_BUILD Register Field Description

Field	Bit	Description
WB_P0_C	[17:15]	Write buffer for peripheral 0 capacity 0x0: no write buffer present 0x1: 16 entries 0x2: 32 entries 0x3: 64 entries 0x4: 128 entries 0x5: 256 entries 0x6: 512 entries 0x7: reserved
WB_P0_E	[18]	Write buffer peripheral 0 write error 0x0: write errors on this interface are not handled by the core. 0x1: write errors on this interface are handled by the core.
WB_P0_I	[20:19]	Write buffer peripheral 0 implementation 0x0: flop-based implementation 0x1: one-cycle SRAM implementation 0x2: two-cycle SRAM implementation 0x3: reserved
WB_P0_O	[21]	Write buffer peripheral 0 outstanding 0x0: 8 write transactions are outstanding on the bus. 0x1: 16 write transactions are outstanding on the bus.
WB_P1_C	[24:22]	Write buffer peripheral 1 capacity 0x0: no write buffer present 0x1: 16 entries 0x2: 32 entries 0x3: 64 entries 0x4: 128 entries 0x5: 256 entries 0x6: 512 entries 0x7: reserved
WB_P1_E	[25]	Write buffer peripheral 1 write error 0x0: write errors on this interface are not handled by the core. 0x1: write errors on this interface are handled by the core.

Table 23-4 WB_BUILD Register Field Description

Field	Bit	Description
WB_P1_I	[27:26]	Write buffer peripheral 1 implementation 0x0: flop-based implementation 0x1: one-cycle SRAM implementation 0x2: two-cycle SRAM implementation 0x3: Reserved
WB_P1_O	[28]	Write buffer peripheral 1 outstanding 0x0: 8 write transactions are outstanding on the bus. 0x1: 16 write transactions are outstanding on the bus.

24

Branch Prediction Unit

24.1 Branch Predication Unit Introduction

The processor uses sophisticated branch-prediction algorithms to minimize the penalty for branches in the code. The resources and parameters in the Branch Prediction Unit (BPU) are configurable to meet the needs of target applications and performance requirements.

24.2 Branch Predication Unit Register Interface

Table 24-1 BPU Registers

Address	Auxiliary Register Name	Description
0x480	BPU_FLUSH	Flush and initialize branch predictor register
0x481	BPU_CTRL	BPU control register
0x482	BC_RAM_ADDR	Branch Cache RAM Address
0x483	BC_RAM_INFO	Branch Cache RAM Information
0x484	BC_RAM_BTA	Branch Cache RAM Branch Target Address
0x485	PT_RAM_ADDR	Prediction Table RAM Address (GShare Index)
0x486	PT_RAM_DATA	Prediction Table RAM Data (Predictions)
0xC0	BPU_BUILD	BPU build configuration register

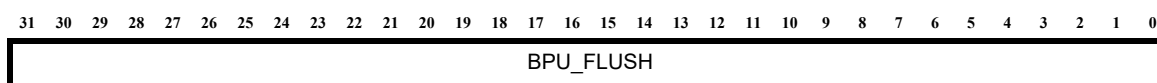
24.2.1 Flush and Initialize Branch Predictor Register, BPU_FLUSH

Address: 0x480

Access: W

Reset: 0x0000_0000

Figure 24-1 BPU_FLUSH Register



The BPU_FLUSH register is used to start an initialization sequence of the branch cache, prediction tables, and prediction state. This initialization sequence flushes all state by cycling through all locations in the branch cache, prediction tables, history registers and setting them to their initial state (prediction taken initial value is 1, and all other BPU RAMs values are 0). The return stack is also flushed.

While the initialization sequence is in progress, no instructions are fetched or issued by the Instruction Fetch Unit.

The BPU_FLUSH is a serializing operation, which must be followed by a restart at the instruction address following the BPU_FLUSH instruction.

Writing any value to BPU_FLUSH starts the initialization sequence.

Writes to the BPU_FLUSH register can be encoded as a 32-bit instruction using the following code:

```
SR 0x480, [0x480]
```

This register may only be written in kernel mode and otherwise triggers a privilege violation exception.



Note

Do not write to the BPU_FLUSH register in a branch delay slot.

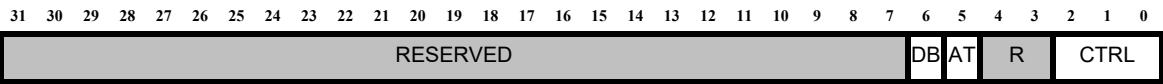
24.2.2 BPU Control Register, BPU_CTRL

Address: 0x481

Access: RW

Reset 0x0

Figure 24-2 BPU_CTRL Register



The BPU_CTRL register is used to dynamically enable branch prediction tables, and prediction state. This register does not affect the unconditional branches. The unconditional branches are always predicated taken.

Access to the BPU_CTRL is a serializing operation, which must be followed by a restart at the instruction address following the instruction. Every write to BPU_CTRL triggers a pipeline flush of the processor that restarts execution with the updated operating mode of the BPU.

Table 24-2 BPU_CTRL Field Description

Field	Bits	Reset	Description
CTRL[2:0]	[2:0]	0	0 Reset value. Indicates the normal operation with branch prediction enabled.
			1 Predict taken. Predict all conditional and unconditional branch or jump instructions in the branch cache to be taken. The state of the prediction tables and global history register is frozen. The branch cache and return stack are still updated. Conditional or unconditional jumps or branches that are not in the branch cache are not predicted and are treated as branches that are not taken. When these are mis-predicted, the branch cache and return stack (for a subroutine call or return) are updated to enable to predict them later. A write to the BPU_FLUSH register resets and initializes the state of the BPU.

Table 24-2 BPU_CTRL Field Description

Field	Bits	Reset	Description
			2 For branch or jump instructions that are in the branch cache: predict all conditional branches as not taken and predict all unconditional branches as taken. The state of the prediction tables and global history register is frozen. The branch cache and return stack are still updated. Conditional or unconditional jumps or branches that are not in the branch cache are not predicted and are treated as branches that are not taken. When these are mis-predicted, the branch cache and return stack (for a subroutine call or return) are updated to enable to predict them later. A write to the <code>BPU_FLUSH</code> register resets and initializes the state of the BPU.
			3 Predict taken with freeze. Predict all unconditional or conditional branches to be taken, if they are in the branch cache. The state of the prediction tables and global history register is frozen. The state of the branch cache is also frozen, including way prediction. However, error branches are purged from the branch cache when they occur. The state of the return stack is updated by predicted subroutine calls and returns. Conditional or unconditional jumps or branches that are not in the branch cache are not predicted and are treated as branches that are not taken. When these jumps or branches are mis-predicted, the BPU state is not updated and they are mis-predicted again when encountered again later. A write to the <code>BPU_FLUSH</code> register resets and initializes the state of the BPU.
			4 Predict not taken with freeze. For branch or jump instructions that are in the branch cache: predict all conditional branches as not taken and predict all unconditional branches as taken. The state of the prediction tables and global history register is frozen. The state of the branch cache is also frozen, including way prediction. However, error branches are purged from the branch cache when they occur. The state of the return stack is updated by predicted unconditional subroutine calls and returns. Conditional or unconditional jumps or branches that are not in the branch cache are not predicted and are treated as branches that are not taken. When these jumps or branches are mis-predicted, the BPU state is not updated and they are mis-predicted again when encountered again later. A write to the <code>BPU_FLUSH</code> register resets and initializes the state of the BPU.
			5 Reserved

Table 24-2 BPU_CTRL Field Description

Field	Bits	Reset	Description
			6 Freeze and do not predict. The state of the prediction tables and global history register is frozen. The state of the branch cache and return stack is also frozen, including way prediction. Error branches also remain in the branch cache. Conditional or unconditional jumps or branches are not predicted and are treated as branches that are not taken. When these branches or jumps are mis-predicted, the BPU state is not updated and they are mis-predicted again when encountered again later. A write to the BPU_FLUSH register resets and initializes the state of the BPU.
			7 Predict with freeze. Predict branches that are in the branch cache with the regular prediction method. The state of the prediction tables, global history register and return stack is still updated for predicted branches. The state of the branch cache, including way prediction, is frozen and no new branches are entered into the branch cache. Error branches are purged from the branch cache when they occur. Conditional or unconditional jumps or branches that are not in the branch cache are not predicted and are treated as branches that are not taken. When these are mis-predicted, the BPU state is not updated and they are mis-predicted again when encountered again later. A write to the BPU_FLUSH register resets and initializes the state of the BPU.
AT	[5]	0	Use this bit to control the BPU debug access control. To use this bit, you must enable the BPU debug access feature: BPU_CTRL[DB] == 1. This bit is ignored when BPU_CTRL[DB] == 0. ■ 0x0: BPU debug direct access; direct access to BPU RAMs (branch cache and prediction table RAMs, most recent BPU computed GShare Index). ■ 0x1: BPU debug controlled access; controlled access to BPU RAMs using current predicted branch (branch index, branch info, BTA, current GShare Index, predictions).
DB	[6]	0	Use this bit to enable or disable BPU debug access. Debug access must be enabled only when debugging BPU. At all other times this bit must be cleared. ■ 0x0: Disable BPU debug access ■ 0x1: Enable BPU debug access
RESERVED	[29:2]	0	Reserved. Read-as-zero; ignored on write.

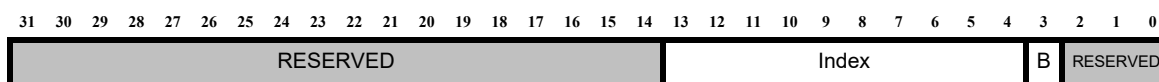
24.2.3 Branch Cache RAM Address Register, BC_RAM_ADDR

Address: 0x482

Access: RW

Reset 0x0000_0000

Figure 24-3 BC_RAM_ADDR Register



Use this register to access the branch cache. The effects of reading and writing to this register are summarized in [Table 24-4](#).

Table 24-3 BC_RAM_ADDR Field Description

Field	Bit	Reset	Description
RESERVED	[2:0]	0	Reserved. Read-as-zero; ignored on write.
B	[3]	0	Specifies the BPU cache bank to read or write. The branch cache has two banks.
Index	[13:4]	0	Specifies the branch cache index. Branch cache MSB bit is used when <code>br_bc_entries==2048</code> . The index range is <code>[n:4]</code> , where $n = \log_2(\text{br\_bc\_entries}/2) + \log_2(\text{br\_block\_size}) - 1$.
RESERVED	[31:14]	0	Reserved. Read-as-zero; ignored on write.

When debug access to BPU is enabled (`BPU_CTRL[DB] == 1`)

Table 24-4 LR and SR to BC_RAM_ADDR Description

Field	LR	SR
<code>BPU_CTRL[AT] == 0</code>	Returns the register value.	1. Updates <code>BC_RAM_ADDR</code> with SR data. 2. Triggers a read from branch cache (RAM) Line. 3. <code>BC_RAM_ADDR</code> value is used as RAM address. 4. Branch Cache auxiliary registers (<code>BC_RAM_INFO</code> and <code>BC_RAM_BTA</code>) are updated with branch cache RAM data.

Table 24-4 LR and SR to BC_RAM_ADDR Description

Field	LR	SR
BPU_CTRL [AT] == 1	Returns the contents of the respective register. When the BPU reads the branch cache RAM to make a prediction, the BC_RAM_ADDR, BC_RAM_INFO, and BC_RAM_DATA registers are updated.	Updates the register with SR data.

When debug access to BPU is disabled (BPU_CTRL [DB] == 0)

- LR returns the register value. LR does not trigger any BPU debug action.
- SR updates register value with SR data. SR does not trigger any BPU debug action.

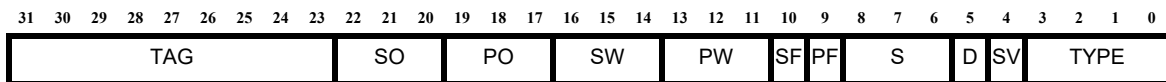
24.2.4 Branch Cache Information Register, BC_RAM_INFO

Address: 0x483

Access: RW

Reset 0x0000_0000

Figure 24-4 BC_RAM_INFO Register



This register provides access to branch cache information. The effects of reading and writing to this register are described in [Table 24-7](#) on page [1096](#).

Table 24-5 BC_RAM_INFO Field Description

Field	Bit	Reset	Description
TYPE	[3:0]	0	Branch type of the primary branch. For more information, see Table 24-6 on page 1095 . When there is no valid primary branch, this field is set to 0.
SV	[4]	0	Secondary prediction validity. 0x0: No secondary branch is stored in the branch cache entry. 0x1: A secondary branch is stored in the branch cache entry.
D	[5]	0	Branch has delay slot 0x0: Branch instruction does not have a delay slot. 0x1: Branch instruction has a delay slot.
S	[8:6]	0	Encodes the size of the primary branch instruction and any potential delay slot instruction and any long immediate operand in the instruction. This size is used to calculate the return address for a subroutine call. The size is in bytes.
PF	[9]	0	Primary branch overflow. 0x0: Next fetch block^a need not be fetched. 0x1: Indicates that the next fetch block must still be fetched for this branch instruction before the branch can be taken, because the tail of the branch instruction or its delay slot instruction straddles a fetch block boundary and spills into the next fetch block.
SF	[10]	0	Secondary Branch Overflow. 0x0: Next fetch block need not be fetched. 0x1: Next fetch block must still be fetched for this branch instruction before the branch can be taken, because the tail of the branch instruction or its delay slot instruction straddles a fetch block boundary and spills into the next fetch block.

Table 24-5 BC_RAM_INFO Field Description

Field	Bit	Reset	Description
PW	[13:11]	0	Primary way (maximum eight-way). Way prediction for the BTA (branch target address) of the primary branch. A four-way set-associative instruction cache has a 2-bit <i>secondary way</i> (SW) field.
SW	[16:14]	0	Secondary way (maximum eight-way). Way prediction for the BTA of the secondary branch. A four-way set-associative instruction cache has a 2-bit <i>secondary way</i> (SW) field.
PO	[19:17]	0	Primary branch offset. Address offset (one of four possible instruction locations) of the primary branch in the fetch block: <code>offset = PC[2:1]</code>
SO	[22:20]	0	Secondary branch offset. Address offset (one of four possible instruction locations) of the primary branch in the fetch block: <code>offset = PC[2:1]</code>
TAG	[31:23]	0	Branch address tag All tag bits are accessible when tag size (<code>br_bc_tag_size</code> and <code>br_bc_full_tag</code>) is less than 9 bits. When tag size is greater than 9 bits, only the least significant bits are accessible. Number of tag bits is 20 when <code>br_bc_full_tag==true</code> . Number of tag bits is equal to <code>br_bc_tag_size</code> when <code>br_bc_full_tag==false</code> . The tag is common for both primary and secondary branches.

- a. The instruction fetch unit (IFU) uses a dual-bank memory configurations, each providing a 64-bit fetch block aligned on a 64-bit address boundary. Thus, the instruction fetch block width is 128 bits. Processors with a vector unit have a 128-bit instruction fetch block. Processors without a vector DSP unit have a 64-bit instruction fetch block. Instruction fetch and branch prediction are based on fetch blocks, blocks of multiple unaligned variable length instructions in memory. A fetch block has an entry point where program execution enters and an exit point where program execution leaves the fetch block. When program execution is purely sequential the entry point is the start of the fetch block and the exit point is the end of the fetch block and continuing into the next sequential fetch block. However, in the presence of branches and other changes in program flow an entry point (a branch target) can be in the middle of a fetch block and an exit point (a taken branch) can also be in the middle of a fetch block. An instruction run is a sequence of instructions from an entry point in one fetch block to an exit point in the same or a sequential fetch block, passing by not taken branches along the way.

Table 24-6 BPU Branch Types

Value	Name	Description	Opcodes
0x0	BR_NOT_PREDICTED	Any instruction other than a valid predicted branch	All opcodes except the ones listed against values 0x1 to 0x7 in this table.
0x1	BR_EI_S	EI_S instruction	EI_S
0x2	BR_CONDITIONAL	Conditional and predicted branch	BBIT0, BBIT1, Bcc, BRcc, Jcc, Bcc_S, BREQ_S, BRNE_S

Table 24-6 BPU Branch Types

Value	Name	Description	Opcodes
0x3	BR_UNCONDITIONAL	Unconditional branch (always taken)	B, BI, BIH, B_S, J_S
0x4	BR_COND_CALL	Conditional subroutine call	JLcc, BLcc
0x5	BR_CALL	Unconditional subroutine call	JL, JL_S, BL_S, JLI_S
0x6	BR_COND_RETURN	Conditional subroutine return	Jcc [blink], JEQ_S [blink], JNE_S [blink]
0x7	BR_RETURN	Unconditional subroutine return	J [blink], J_S [blink], LEAVE_S (if u[6]==1)
0x8 to 0xF	RESERVED		

When debug access to BPU is enabled (BPU_CTRL[DB] == 1)

Table 24-7 LR and SR to BC_RAM_INFO Description

Field	LR	SR
BPU_CTRL[AT] == 0	Returns the register value.	1. Updates BC_RAM_INFO with SR data. 2. Triggers a write to the branch cache (RAM) Line. 3. BC_RAM_ADDR value is used as RAM address. 4. Branch cache line is updated with the branch cache auxiliary registers (BC_RAM_INFO and BC_RAM_BTA) as RAM data.
BPU_CTRL[AT] == 1	Returns the contents of the respective register. When the BPU reads the branch cache RAM to make a prediction, the BC_RAM_ADDR, BC_RAM_INFO, and BC_RAM_DATA registers are updated.	Updates the register with SR data.

When debug access to BPU is disabled (BPU_CTRL[DB] == 0)

- LR returns the register value. LR does not trigger any BPU debug action.
- SR updates register value with SR data. SR does not trigger any BPU debug action.

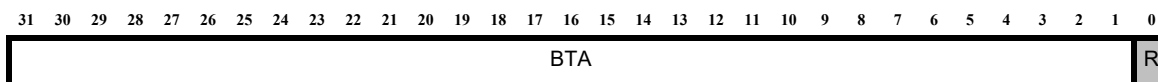
24.2.5 Branch Cache Branch Target Address Register, BC_RAM_BTA

Address: 0x484

Access: RW

Reset 0x0000_0000

Figure 24-5 BC_RAM_BTA Register



This register provides access to branch cache RAM that stores the branch target address. The effects of reading and writing to this register are described in [Table 24-9](#).

Table 24-8 BC_RAM_BTA Field Description

Field	Bit	Reset	Description
RESERVED	[0]	0	Reserved. Read-as-zero; ignored on write.
BTA	[31:1]	0	Branch cache branch target address

When debug access to BPU is enabled (BPU_CTRL[DB] == 1)

Table 24-9 LR and SR to BC_RAM_BTA Description

Field	LR	SR
BPU_CTRL[AT] == 0	Returns the register value.	1. Updates BC_RAM_BTA with SR data. 2. Triggers a write to the branch cache (RAM) Line. 3. BC_RAM_ADDR value is used as RAM address. 4. Branch cache line is updated with the branch cache auxiliary registers (BC_RAM_INFO and BC_RAM_DATA) as RAM data.
BPU_CTRL[AT] == 1	Returns the contents of the respective register. When the BPU reads the branch cache RAM to make a prediction, the BC_RAM_ADDR, BC_RAM_INFO, and BC_RAM_DATA registers are updated.	Updates the register with SR data.

When debug access to BPU is disabled (`BPU_CTRL[DB] == 0`)

- LR returns the register value. LR does not trigger any BPU debug action.
- SR updates register value with SR data. SR does not trigger any BPU debug action.

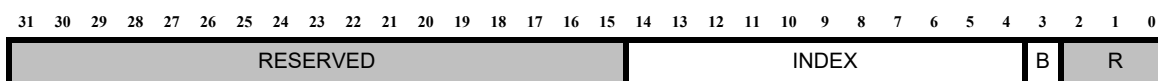
24.2.6 Prediction Table Address Register, PT_RAM_ADDR

Address: 0x485

Access: RW

Reset 0x0000_0000

Figure 24-6 PT_RAM_ADDR Register



This register provides access to the prediction table RAM address. The effects of reading and writing to this register are described in [Table 24-11](#).

Table 24-10 PT_RAM_ADDR Bit Field Description

Field	Bits	Reset	Description
RESERVED	[2:0]	0	Reserved. Read-as-zero; ignored on write.
B	[3]	0	Specifies the prediction table bank to access.
Index	[11:4]	0	Specifies the index of a line in the prediction table. The MSB bit is used when <code>br_pt_entries=16384</code> . LSB is determined by the fetch block size. Number of actual bits used is determined by the <code>br_pt_entries</code> option configuration. The index range is [n: 4], where $n = \log_2 (\text{br\_pt\_entries}/2) + 1$ when (FETCH_BLOCK == 128) $n = \log_2 (\text{br\_pt\_entries}/4) + 1$ when (FETCH_BLOCK == 64).
RESERVED	[31:15]	0	Reserved. Read-as-zero; ignored on write.

When debug access to BPU is enabled (`BPU_CTRL[DB] == 1`)

Table 24-11 LR and SR to PT_RAM_ADDR

Access Type	LR	SR
<code>BPU_CTRL[AT] == 0</code>	Returns the register value.	1. Updates PT_RAM_ADDR with SR data. 2. Triggers a read from the prediction table (RAM) line. 3. PT_RAM_ADDR value is used as RAM address. 4. Prediction table data register (PT_RAM_DATA) is updated with RAM data.

Table 24-11 LR and SR to PT_RAM_ADDR

Access Type	LR	SR
BPU_CTRL[AT] == 1	Returns the contents of the respective register. When the BPU reads the prediction table RAM to make a prediction, the PT_RAM_ADDR and PT_RAM_DATA registers are updated.	Updates the register with SR data.

When debug access to BPU is disabled (BPU_CTRL[DB] == 0)

- LR returns the register value. LR does not trigger any BPU debug action.
- SR updates register value with SR data. SR does not trigger any BPU debug action.

24.2.7 Prediction Table DATA Register, PT_RAM_DATA

Address: 0x486

Access: RW

Reset 0x0000_0000

Figure 24-7 PT_RAM_DATA Register when FETCH_BLOCK==128

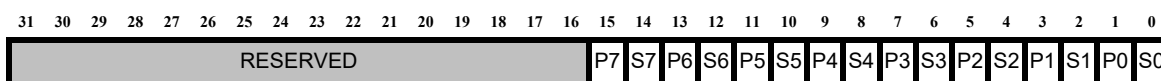
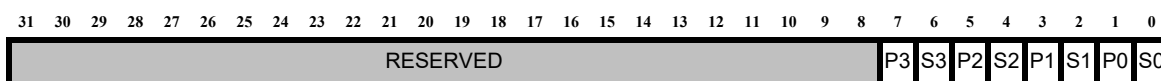


Figure 24-8 PT_RAM_DATA Register when FETCH_BLOCK==64



This register provides access to prediction table data. The effects of reading and writing to this register are described in [Table 24-13](#) on page 1102.

Table 24-12 PT_RAM_DATA Bit Field Description

Field	Reset	Description
S_n	0	Prediction strength of instruction n . Where n is the byte offset of an instruction in an instruction fetch block. 0x0: Weak 0x1: Strong
P_n	0	Prediction taken of instruction n . Where n is the byte offset shifted right by 1 of an instruction in an instruction fetch block. 0x0: branch is not taken 0x1: branch is taken
RESERVED	0	Reserved. Read-as-zero; ignored on write.

When debug access to BPU is enabled (BPU_CTRL[DB] == 1)**Table 24-13 LR and SR to PT_RAM_DATA**

Access Type	LR	SR
BPU_CTRL[AT] == 0	Returns the register value.	1. Updates PT_RAM_DATA with SR data. 2. Triggers a write to the prediction table (RAM) Line. 3. PT_RAM_ADDR value is used as RAM address. 4. Prediction table line is updated with register data (PT_RAM_DATA).
BPU_CTRL[AT] == 1	Returns the contents of the respective register. When the BPU reads the prediction table RAM to make a prediction, the PT_RAM_ADDR and PT_RAM_DATA registers are updated.	Updates the register with SR data.

When debug access to BPU is disabled (BPU_CTRL[DB] == 0)

- LR returns the register value. LR does not trigger any BPU debug action.
- SR updates register value with SR data. SR does not trigger any BPU debug action.

24.2.8 Branch Prediction Unit Configuration Register, BPU_BUILD

Address: 0xC0

Access: R

Figure 24-9 BPU_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				BDB		FBE		TQE		TS				FT		RSE		PTE			BCE			VERSION							

Table 24-14 lists the field descriptions.

Table 24-14 BPU_BUILD Field Descriptions

Field	Bits	Description
VERSION	[7:0]	Branch Prediction Unit version 0x05: First version of branch prediction unit in ARCV2-based processors. 0x06: Current version. Includes enhanced BPU control options (see BPU_CTRL[2:0] = 3 4 6 7).
BCE	[10:8]	Number of Branch Cache (BC) entries 0x0: 256 0x1: 512 0x2: 1 K 0x3: 2 K All other values reserved.
PTE	[13:11]	Number of Prediction Table (PT) entries 0x0: 2 K 0x1: 4 K 0x2: 8 K 0x3: 16 K All other values reserved.
RSE	[15:14]	Number of Return-Stack (RS) entries 0x0: 4 0x1: 8 All other values reserved.

Table 24-14 BPU_BUILD Field Descriptions (Continued)

Field	Bits	Description
FT	[16]	Use full tag in branch cache 0x0: Use partial tag in branch cache. If a partial tag is used, field TS indicates the size of that partial tag. If a full-size-tag is used, field TS is ignored. 0x1: Use full size tag in branch cache.
TS	[21:17]	Size of branch cache tag (bits) - If FT == 0, this field reflects the user selected tag size (BR_BC_TAG_SIZE). - If FT == 1, this field reflects the size of the full tag, determined by the size of the branch cache given by the following formula: - With the Vector DSP, $BPU\_BUILD.TS = \log_2\text{ceil}(BPU\_BUILD.BCE) + 10$ - Without vector DSP, $BPU\_BUILD.TS = \log_2\text{ceil}(BPU\_BUILD.BCE) + 11$
TQE	[23:22]	Number of entries for Top of Stack (TOS) queue 0x1: 2 0x2: 5 0x3: 8
FBE	[25:24]	Number of eight-byte entries in IFU buffer 0x0: 1 0x1: 2 All other values reserved.
BDB	[27:26]	Indicates support for BPU debug capability: auxiliary access to BPU memories 0x0: No support for debugging BPU 0x1: Support for debugging BPU 0x2 - 0x3: Reserved
RESERVED	[31:28]	Reserved. Read-as-zero; ignored on write.

24.3 Debugging BPU

The BPU uses two RAMs: Branch Cache (BC) and Prediction Table (PT) to store information about branches and their prediction. The BPU also includes a Global History Register (GHR) and a return stack address to predict branch direction and its branch target. When you configure the BPU debug feature in the processor, you can access these RAMs, GHR, and branch target information using the BPU debug auxiliary registers.

24.3.1 Configuring BPU Debug Feature

To configure the BPU debug feature in the processor, you must use the `-br_dbg_option` configuration option.

24.3.2 Enabling BPU Debug Access

To enable BPU debugging, you must set `BPU_CTRL [DB] == 1`.

24.3.3 Best Practices to Follow When Debugging BPU

- To use this feature, you must understand the BPU GShare algorithm.
- Before you debug BPU, ensure that the BPU is not configured in the Freeze and do not predict mode. If the BPU is not in this mode, reading the BPU RAMs returns indeterminate values.
- When the processor is unresponsive, you can debug the BPU only if you have configured the NMI (non-maskable imprecise exception) feature in the processor.
- To perform normal BPU operations, ensure that the `BPU_CTRL [DB]` bit is cleared to avoid any unintended corruption of BPU functionality.
- You can access the BPU debug data using the external host debugging mode or through the core (after setting the BPU in freeze-and-do-not-predict mode (`BPU_CTRL [1 : 0] == 0x6`)).
- BPU debug controlled access mode must be used only with single-step debugging.

24.3.4 Methods to Access the BPU Auxiliary Registers

The BPU debug feature allows you to access the BPU RAMs, GHR, and return stack using the following methods:

- BPU controlled access mode: Use this mode to access the BPU debug information related to the most recent branch. Use this mode with the debugger's single-stepping instruction feature. Set `BPU_CTRL [DB] == 1` to program the BPU in this mode. In this mode, when the BPU reads the branch cache RAM to make a prediction, the `BC_RAM_ADDR`, `BC_RAM_INFO`, and `BC_RAM_DATA` registers are updated. Similarly, when the BPU reads the prediction table to make a prediction, the `PT_RAM_ADDR` and `PT_RAM_DATA` registers are updated.
- Direct access mode: In this mode, you can directly access the BPU debug information of all branches stored in the BPU RAMs. Set `BPU_CTRL [AT] == 0` to program the BPU in this mode.

Use the following auxiliary registers to access the BPU debug data:

Table 24-15 BPU Debug Auxiliary Registers

Address	Auxiliary Register Name
0x482	<code>BC_RAM_ADDR</code>
0x483	<code>BC_RAM_INFO</code>
0x484	<code>BC_RAM_BTA</code>
0x485	<code>PT_RAM_ADDR</code>
0x486	<code>PT_RAM_DATA</code>

24.3.5 What Data Can be Accessed

See the “[BC_RAM_INFO](#)”, “[BC_RAM_BTA](#)”, and “[PT_RAM_DATA](#)” registers for the debug information that is available in the BPU.

24.3.6 How to Dump Branch Cache and Prediction Table Data

- A write (debugger write or SR operation) to the `BC_RAM_ADDR` register updates the branch information in the `BC_RAM_INFO` register and the branch target address in the `BC_RAM_BTA` register.
- A write (debugger write or SR operation) to the `PT_RAM_ADDR` register updates the `PT_RAM_ADDR` register with the GHR, and `PT_RAM_DATA` with the contents of prediction table RAM.
- A write (debugger write or SR operation) to the `BC_RAM_INFO` or `BC_RAM_BTA` register triggers an update of the branch cache RAM location, pointed by the `BC_RAM_ADDR` register, with contents of the `BC_RAM_INFO` and `BC_RAM_BTA` registers.
- A write (debugger write or SR operation) to the `PT_RAM_DATA` register triggers an update of prediction table RAM location, pointed by the `PT_RAM_ADDR` register, with contents of the `PT_RAM_DATA` register.

24.3.7 Usage Examples

24.3.7.1 BPU Debug Direct Read Access Mode

```
DES_PC equ <Desired PC>
RAM_ST_ADDR equ <BPU RAM Debug Start Address>
RAM_EN_ADDR equ <BPU RAM Debug End Address>

_bpu_int_excpn_hndl:
;; Setup BPU Direct Access Mode
lr      r4, [BPU_CTRL]    ; Access BPU control register
mov     r3, 0xFFFF_FFF8  ; [2:0] == 0x0
and     r4, r4, r3        ; Clear BPU_CTRL [2:0]
or      r4, r4, 0x6       ; Freeze Predictions BPU_CTRL [2:0] (ctrl_flag = FP, 0x6)
bclr   r4, r4, at_flag    ; Clear AT Flag in BPU_CTRL (at_flag = AT, 0x20)
bset   r4, r4, db_flag    ; Set DB Flag in BPU_CTRL (db_flag = DB, 0x40)
sr      r4, [BPU_CTRL]    ; program BPU debug 'Direct Access Mode'
mov     r4, RAM_ST_ADDR   ; Access BPU control register
mov     r5, RAM_EN_ADDR   ; Access BPU control register

_bc_ram:
mov     r6, r4            ; program index to RAM_ST_ADDR
_bc_loop:
sr      r6, [BC_RAM_ADDR] ; program BC RAM ADDR
lr      r7, [BC_RAM_INFO] ; program read BC RAM INFO
lr      r7, [BC_RAM_BTA]  ; program read BC RAM BTA
cmp     r6, r5            ; program check index at end address
bneq.d _bc_loop          ; program loop to next RAM address
add     r6, r6, 0x4       ; program increment RAM address

_pt_ram:
mov     r6, r4            ; program index to RAM_ST_ADDR
```

```
_pt_loop:
sr      r6, [PT_RAM_ADDR]; program PT RAM ADDR
lr      r7, [PT_RAM_DATA]; program read PT RAM DATA
cmp     r6, r5           ; program check index at end address
bneq.d _pt_loop         ; program loop to next RAM address
add     r6, r6, 0x4      ; program increment RAM address
```

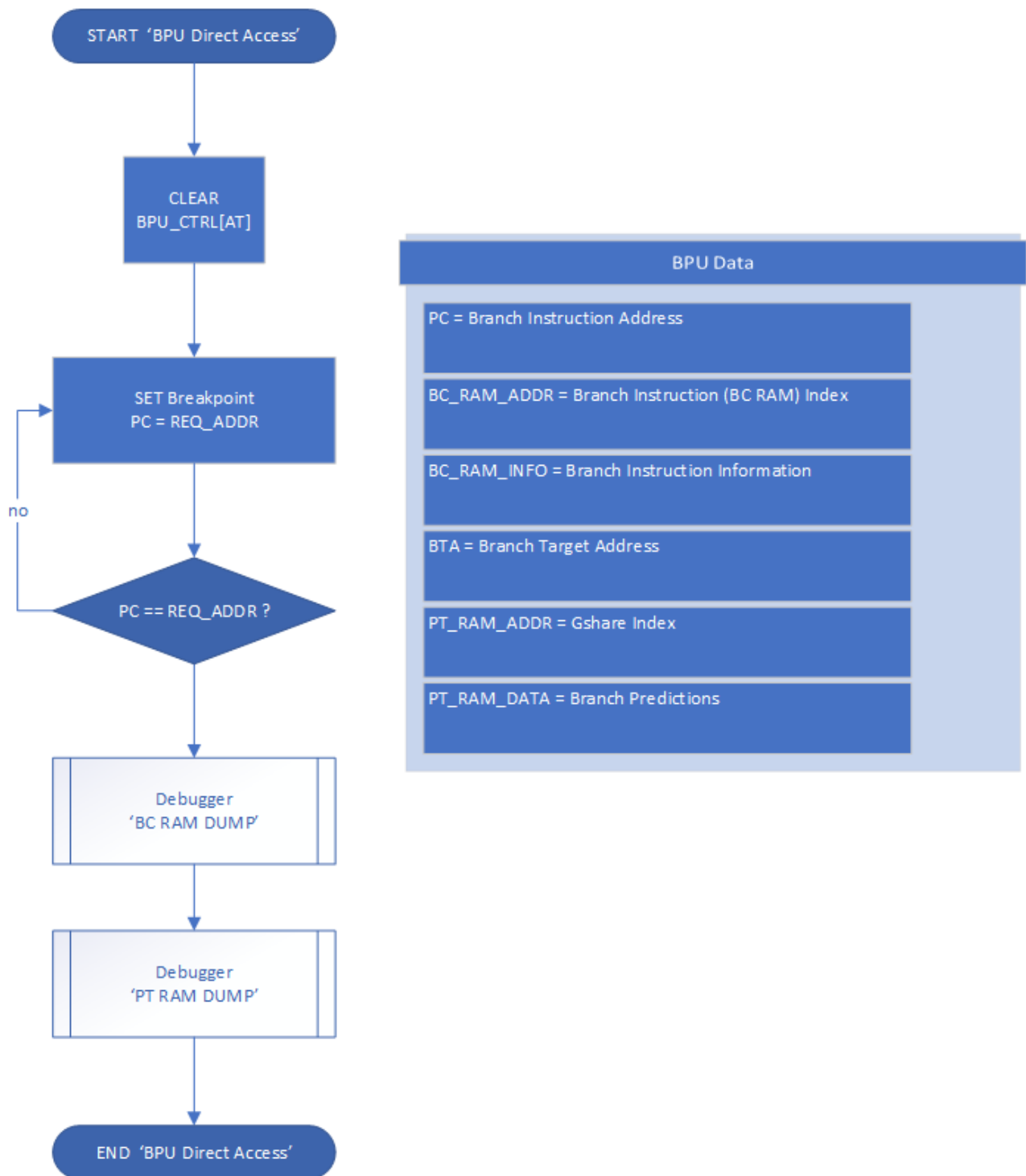
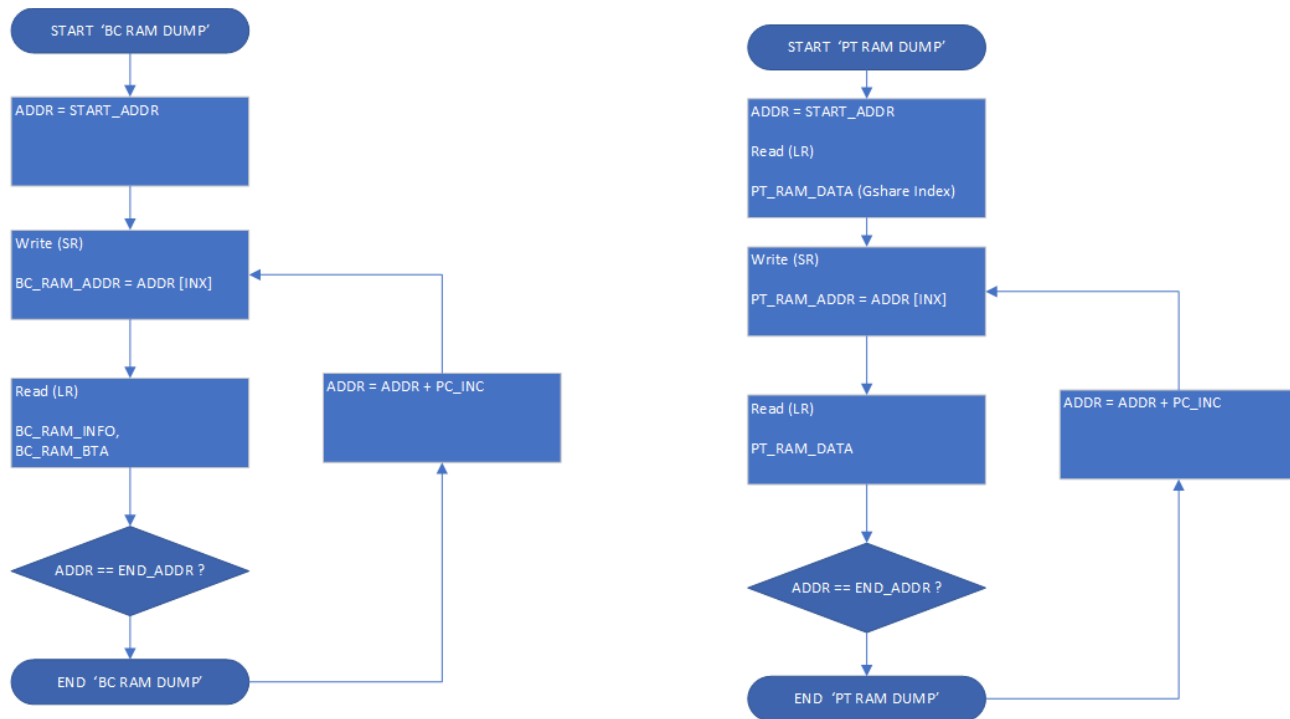
Figure 24-10 BPU RAM Direct Access

Figure 24-11 BPU BC and PT RAM DUMP**24.3.7.2 Read Access through BPU Debug Controlled Access Mode**

```
;; Debugger single step access
```

```
DES_PC equ <Desired PC>
```

```
RAM_ST_ADDR equ <BPU RAM Debug Start Address>
```

```
RAM_EN_ADDR equ <BPU RAM Debug End Address>
```

```
;; Debugger set breakpoint on desired PC to dump BPU RAMs
```

```
brk DES_PC ; BPU Debug desired PC
```

```
_start:
```

```
;; Setup BPU Controlled Access Mode (through debugger)
```

```
lr r4, [BPU_CTRL] ; Access BPU control register
```

```
bset r4, r4, at_flag ; Set AT Flag in BPU_CTRL (at_flag = AT, 0x20)
```

```
bset r4, r4, db_flag ; Set DB Flag in BPU_CTRL (db_flag = DB, 0x40)
```

```
sr r4, [BPU_CTRL] ; program BPU debug 'Controller Access Mode'
```

```
mov r4, RAM_ST_ADDR ; Access BPU control register
```

```
mov r5, RAM_EN_ADDR ; Access BPU control register
```

```
_bp_freeze:
```

```
lr r10, [BPU_CTRL] ; Access BPU control register
```

```
mov r9, 0xFFFF_FFF8 ; [2:0] == 0x0
```

```
and r8, r10, r9 ; Clear BPU_CTRL [2:0]
```

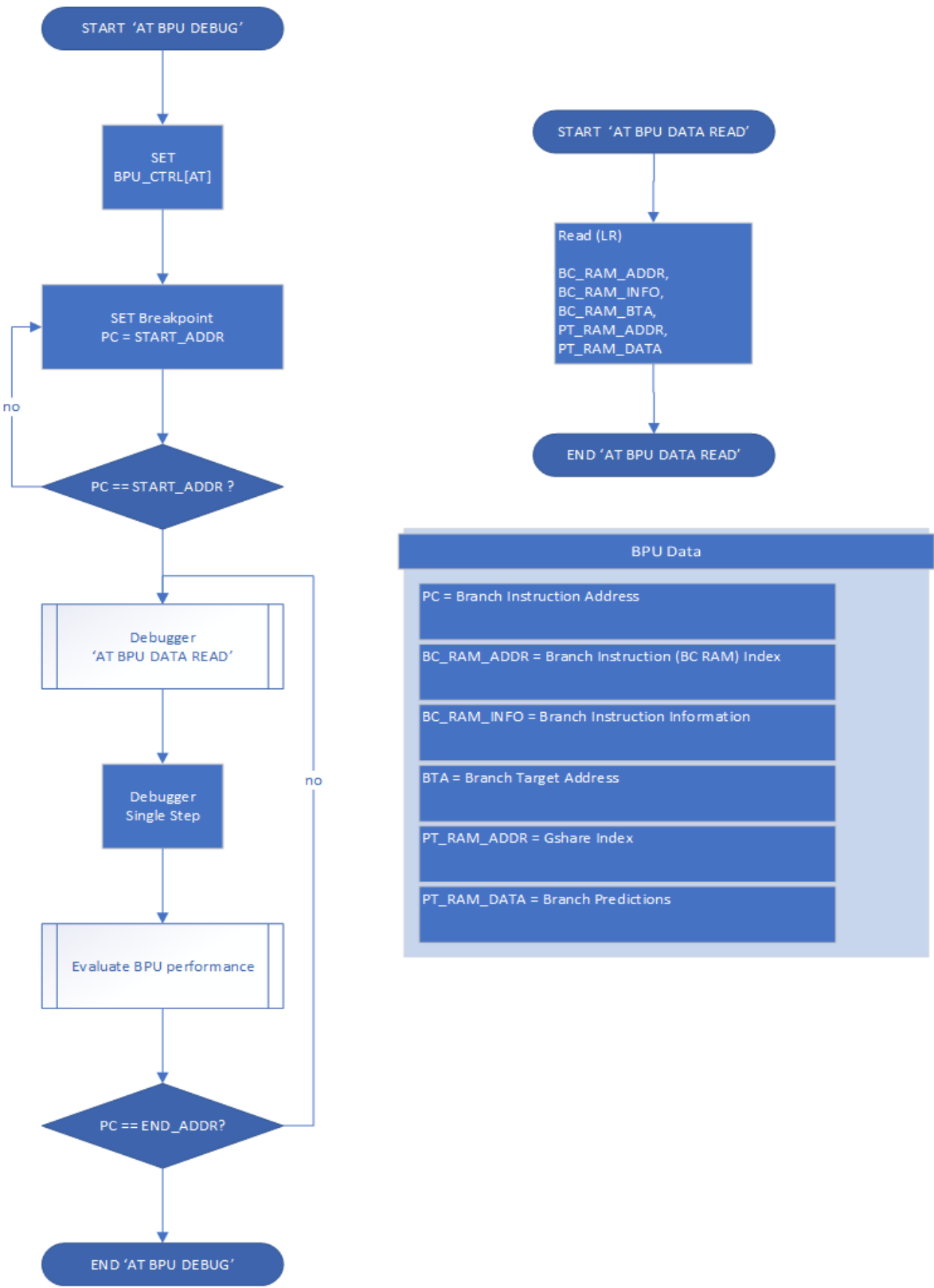
```
or r8, r8, 0x6 ; Freeze Predictions BPU_CTRL [2:0] (ctrl_flag = FP, 0x6)
```

```
sr r8, [BPU_CTRL] ; program BP Freeze
```

```
_bc_ram:
mov     r6, r4           ; program index to RAM_ST_ADDR
sr      r6, [BC_RAM_ADDR] ; program BC RAM ADDR
lr      r7, [BC_RAM_INFO] ; program read BC RAM INFO
lr      r7, [BC_RAM_BTA]  ; program read BC RAM BTA
cmp     r6, r5           ; program check index at end address
bneq.d  _bc_ram          ; program loop to next RAM address
add     r6, r6, 0x4       ; program increment RAM address

_pt_ram:
mov     r6, r4           ; program index to RAM_ST_ADDR
sr      r6, [PT_RAM_ADDR] ; program PT RAM ADDR
lr      r7, [PT_RAM_DATA] ; program read PT RAM DATA
cmp     r6, r5           ; program check index at end address
bneq.d  _pt_ram          ; program loop to next RAM address
add     r6, r6, 0x4       ; program increment RAM address

;; Debugger single step on desired PC to dump BPU RAMs
sr      r8, r10           ; program BP revert to original Prediction settings
step    DES_PC            ; BPU Debug desired PC
cmp     r6, r5           ; program check index at end address
beq.d   _bp_freeze        ; program loop to next RAM address
```



24.3.7.3 BPU Debug Direct Read Access Mode (in Exception or Interrupt Handler)

```

DES_PC equ <Desired PC>
RAM_ST_ADDR equ <BPU RAM Debug Start Address>
RAM_EN_ADDR equ <BPU RAM Debug End Address>

_bpu_int_excptn_hndl:
;; Setup BPU Direct Access Mode
lr      r4, [BPU_CTRL]    ; Access BPU control register
mov     r3, 0xFFFF_FFF8  ; [2:0] == 0x0
and     r4, r4, r3        ; Clear BPU_CTRL [2:0]
or      r4, r4, 0x6       ; Freeze Predictions BPU_CTRL [2:0] (ctrl_flag = FP, 0x6)
bclr   r4, r4, at_flag    ; Clear AT Flag in BPU_CTRL (at_flag = AT, 0x20)
bset   r4, r4, db_flag    ; Set DB Flag in BPU_CTRL (db_flag = DB, 0x40)
sr      r4, [BPU_CTRL]    ; program BPU debug 'Direct Access Mode'
mov     r4, RAM_ST_ADDR   ; Access BPU control register
mov     r5, RAM_EN_ADDR   ; Access BPU control register

_bc_ram:
mov     r6, r4            ; program index to RAM_ST_ADDR
_bc_loop:
sr      r6, [BC_RAM_ADDR] ; program BC RAM ADDR
lr      r7, [BC_RAM_INFO] ; program read BC RAM INFO
lr      r7, [BC_RAM_BTA]  ; program read BC RAM BTA
cmp     r6, r5            ; program check index at end address
beq.d   _bc_loop          ; program loop to next RAM address
add     r6, r6, 0x4        ; program increment RAM address

_pt_ram:
mov     r6, r4            ; program index to RAM_ST_ADDR
_pt_loop:
sr      r6, [PT_RAM_ADDR] ; program PT RAM ADDR
lr      r7, [PT_RAM_DATA] ; program read PT RAM DATA
cmp     r6, r5            ; program check index at end address
beq.d   _pt_loop          ; program loop to next RAM address
add     r6, r6, 0x4        ; program increment RAM address

```


25

Loop Buffer

25.1 Introduction

The loop buffer is a small level-0 cache for the loop body of a loop demarcated by an LP and LPcc instruction. The loop buffer works with the branch-prediction unit in the core to enter and exit loops, which can generally be done without mispredicting entry and exit of the loop. The loop prediction and loop-buffer operation only work if no branch instructions are taken in the loop body. A taken branch instruction in the loop body raises a misprediction and terminates the loop-buffer mechanism for that loop.

The loop buffer saves power: when instructions are fetched from the loop buffer, the instruction cache, ICCM, and branch predication unit are turned off and their power is saved if the loop body fits in the loop buffer. If the loop body does not fit completely in the loop buffer, only the part of the loop that fits in the loop buffer is cached and the rest of the loop body is retrieved from the instruction cache or ICCM, thereby achieving partial power saving.

When LP_START is in an odd numbered fetch block, the loop buffer leaves one fetch block unused to make sure the tail is aligned on a 16-byte boundary. If the size of the loop buffer is 64 bytes and LP_START is in an odd-numbered fetch block, the loop buffer contains 56 bytes and the tail starts at (LP_START + 56), rounded down to a 16 byte boundary.

25.2 Configuring the Loop Buffer

In ARChitect, specify the following option to configure a loop buffer in the processor:

```
-lpb_num=0 | 2 | 4
```

The loop buffer size (in bytes) is calculated as follows:

```
loop_buffer_size = (Instruction_fetch_block_width) * lpb_num
```

25.3 Programming the Loop Buffer

Loop-buffer operation can be enabled or disabled using the LPB_CTRL register.

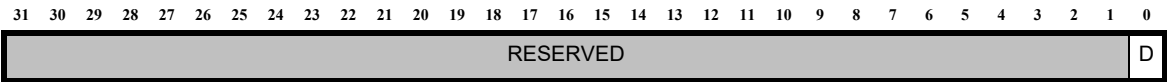
25.3.1 Loop Buffer Control, LPB_CTRL

Address: 0x488

Access: RW

Reset: 0x0

Figure 25-1 LPB_CTRL Register



Use this register to enable or disable loop-buffer operation.



- When the loop buffer is enabled, avoid nested LP instructions when the inner LP instruction has an improper displacement. An improper displacement for an LP instruction is defined as:
 - If the loop body is less than 4 bytes
 - If the displacement is negative ($\text{improper\_displacement} = (\text{lp\_displacement\_r}[11:1] < (\text{align\_pc\_r}[1] + 4))$)
- If you intend to use nested LP instructions with improper displacement, disable the loop buffer.

Table 25-1 LPB_CTRL Field Description

Field	Bit	Reset Value	Description
D	[1]	0	Disable the loop buffer. ■ 0x0: loop-buffer operation is enabled. ■ 0x1: loop-buffer operation is disabled, and the regular instruction cache, ICCM, and BPU operation are enabled.

25.3.2 Loop Buffer Configuration Register, LPB_BUILD

Address: 0xE9

Access: R

Figure 25-2 LPB_BUILD Register



This register specifies the version number of the loop buffer and the loop-buffer size. If you have not configured the loop buffer in your processor, reading this register returns 0, and the loop buffer auxiliary registers are not included in the processor; in this case, reading or writing to the loop buffer auxiliary registers raises an illegal-instruction exception.

Table 25-2 LPB_BUILD Field Description

Field	Bit	Reset Value	Description
VERSION	[7:0]	1	Version number. 0x1: first version of the loop buffer.
LPB_NUM	[15:8]	0	Number of loop buffer entries. 0x0: 0 entries 0x2: 2 entries 0x4: 4 entries. This field reflects the value of the configuration option LPB_NUM.

26

External Host Debugging

26.1 External Host Debugging Introduction

If the configuration includes the interface to an external debug host, additional registers are included in the auxiliary register set. This auxiliary register set includes at least the DEBUG and DEBUGI register, and may also include other registers to support, for example, real-time tracing capabilities. For specific details of advanced debug capabilities, refer to the databook for your ARCV2-based processor.

26.2 Host Debug Register Interface

Table 26-1 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

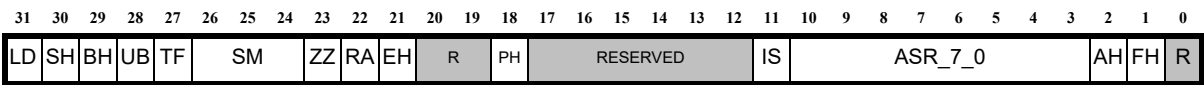
26.2.1 Debug Register, DEBUG

Address: 0x05

Access: RG

Reset: 0x0040_0000

Figure 26-1 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 1129.

The debug register (DEBUG) contains the following bits:

Table 26-2 DEBUG Register Field Description

Field	Bit	Reset	Description
Force halt (FH)	[1]	0	This bit exists in configurations that include ARC Trace. FH is the approved method of stopping the ARCV2-based processor externally by the host. This bit does not have any affect if the ARCV2-based processor is already halted and the host sets this bit. The FH bit is not a mirror of the STATUS register H bit, that is, clearing FH does <i>not</i> start the processor. The FH bit always returns 0 when it is read.
Actionpoint Halt Flag (AH)	[2]	0	The actionpoint halt flag (bit 2), of the DEBUG register, indicates that an actionpoint has halted the processor when the halt flag is set to 1. The AH flag is cleared when the halt flag (H) in the STATUS32 register (see STATUS32) is cleared, for example, by restarting (see Starting) or single-stepping (see Single Instruction Stepping) the ARCV2-based processor. When actionpoints use a software exception, the AH bit to 0. Note: This bit is available only if actionpoints are enabled in the build (has_actionpoints==true). If actionpoints are disabled (has_actionpoints==false), this bit is read as zero and ignored on write.

Table 26-2 DEBUG Register Field Description

Field	Bit	Reset	Description
Actionpoints Status Register field (ASR_7_0)	[10:3]	0	The actionpoints Status Register field (bits [10:3]) of the DEBUG register, indicates which actionpoints have triggered an actionpoint event, independent of whether the actionpoint raises a software exception or a processor halt. Each bit, from ASR0 to ASR7, indicates which actionpoints from 0 to 7 have triggered the actionpoint event. When actionpoints are linked forming pairs or quads, only the master actionpoint of the pair or quad has its ASR bit set when the actionpoint is triggered. The ASR bits are sticky, and Actionpoints trigger only when their corresponding ASR bit transitions from 0 to 1. Therefore, when an Actionpoint condition is met and the ASR bit for the Actionpoint is set to 1, the exception or breakpoint associated with that Actionpoint is triggered only if ASR bit was previously 0. The corresponding ASR bit for an actionpoint is automatically cleared when the host or core writes to any of the actionpoint's associated control registers AP_AMVn, AP_AMMn, or AP_ACn (see Actionpoint Match Value, AP_AMV0, Actionpoint Match Mask, AP_AMM0, and Actionpoint Control, AP_AC0). Note: This field is available only if actionpoints are enabled in the build (has_actionpoints==true). If actionpoints are disabled (has_actionpoints==false), this field is read as zero and ignored on write.
Single instruction step (IS)	[11]	0	Single instruction stepping is provided through the use of the IS bit. If the host sets the IS bit, the ARCV2 core executes one instruction.
RESERVED	[20:19] and [17:12]	0	Reserved. Read-as-zero; ignored on write.
Patch Halt (PH)	[18]	0	This bit is reserved if the ROM patching unit (RPU) is not included in your processor. This bit is cleared on reset. This bit is set to 1 when the RPU triggers a patch breakpoint and halts the core.
External Halt (EH)	[21]	0	The EH bit is set when the processor is halted by external logic. The <code>arc_halt_req_a</code> signal is asserted by external logic that needs to halt the ARCV2-based processor. This signal may be asserted asynchronously with respect to the processor clock. When the processor is ready to halt, it enters the halted state in response to the assertion of <code>arc_halt_req_a</code> , sets the EH bit, and asserts the <code>arc_halt_ack</code> acknowledgment signal. This bit is a status bit, and the host cannot write to this bit.
Reset applied (RA)	[22]	1	RA bit is used by the debug host to determine that a target reset has occurred. The RA bit is set to 1 on a hard reset of the CPU and can be written only by the host.

Table 26-2 DEBUG Register Field Description

Field	Bit	Reset	Description
Sleep state (ZZ)	[23]	0	ZZ bit indicates that the ARCV2-based processor is in the “sleep” state. Use the SLEEP instruction to force the ARCV2-based processor enter the sleep state. This bit is cleared whenever the ARCV2-based processor wakes from sleep state. This bit is a status bit, and the host cannot write to this bit.
Sleep Mode (SM)	[26:24]	0	SM field indicates the sleep mode of the processor. This field controls whether clocks for the internal timers and real-time clock are turned off during sleep state. For more information about the state of internal clocks during sleep state, see Table 26-3 . This bit is a status bit, and the host cannot write to this bit.
Triple Fault (TF)	[27]	0	When this bit is set to 1, it indicates that the processor has halted due to a triple fault. A triple fault occurs when an exception occurs during the fetching of an exception vector for an EV_MachineCheck (double fault) exception.
User-mode breakpoint (UB)	[28]	0	UB bit indicates that BRK is enabled in user mode. This bit is provided to allow an external debugger to debug user-mode tasks. This bit is always set to 0 to ensure that a user-mode task cannot stop the processor by executing a BRK instruction. If actionpoints are enabled, actionpoint halt in user mode requires that the DEBUG.UB bit is set. When transitioning from a kernel mode to user mode, use a SYNC instruction is required (DEBUG.UB cleared) to ensure that all kernel watchpoints are taken before transitioning to the user mode.
Breakpoint halt (BH)	[29]	0	BH bit is set when core is halted by execution of a breakpoint. This bit is cleared when the H bit in the STATUS32 register is cleared, that is, when single-stepping or restarting the ARCV2-based processor from the halted state.
Self halt (SH)	[30]	0	SH indicates that the ARCV2-based processor has halted itself with the FLAG instruction. This bit is cleared whenever the H bit in the STATUS register is cleared, that is, the ARCV2-based processor is running or a single step has been executed.
Load pending bit (LD)	[31]	0	The host (see “The Host” on page 849) 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 26-3 lists the state of various internal clocks during various sleep modes.

Table 26-3 Internal Clock During Sleep Mode

Sleep Mode (DEBUG.SM)	Core	Timers	Real-time Clock	IRQ Resync Clock
0	Disabled	Enabled	Enabled	Disabled unless required
1	Disabled	Disabled	Enabled	Disabled unless required
2	Disabled	Enabled	Disabled	Disabled unless required
3	Disabled	Disabled	Disabled	Disabled unless required
4	Disabled	Disabled	Disabled	Disabled unless required
5	Disabled	Disabled	Disabled	Disabled unless required
6	Disabled	Disabled	Disabled	Disabled unless required
7	Disabled	Disabled	Disabled	Disabled unless required



Note

If the core is configured with a watchdog timer, and the watchdog timer is not configured to be clocked with an external clock, the watchdog timer is also disabled for all sleep modes except sleep mode 0 and 2.

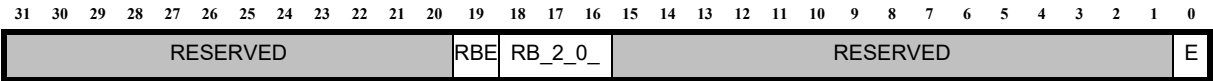
26.2.2 Interrupt and Register Bank Debug Register, DEBUGI

Address: 0x0F

Access: RG

Reset: 0x0000_0000

Figure 26-2 DEBUGI Register



This register allows the debug host an independent selection of which register bank to access when accessing core registers.

You can also use this register through a debug host to enable the exception or interrupt vector break functionality.

Table 26-4 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 completed before the breakpoint is executed.
RESERVED	[15:1]	Reserved. Read-as-zero; ignored on write.

Table 26-4 DEBUGI Register Field Description

Field	Bit	Description
RB_2_0_	[18:16]	Register Bank (present with <code>RGF_NUM_BANKS > 1</code>) Pre-empts STATUS32.RB for all instructions originating from the debug interface when DEBUGI.RBE is set. When DEBUGI.RBE is not set, the existing value STATUS32.RB is used where RBE = bit 19. Following are the field values: ■ 001: Two register banks ■ 010: Three register banks ■ 011: Four register banks ■ 100: Five register banks ■ 101: Six register banks ■ 110: Seven register banks ■ 111: Eight register banks Note: When only when one register bank is configured, this field does not exist. This field width is variable. The bits with value 0 in the MSBs are not implemented. For example, in a two-bank configuration, bits [18:17] do not exist in this register. When the number of configurable register banks is a power of two, the STATUS32.RB field can never point to an unimplemented register bank. When the number of configurable register banks is not a power of two (3, 5, 6 or 7), the STATUS32.RB field value is constrained to never point to a register bank that does not exist. On an auxiliary write or other update of the STATUS32 register RB field, if the update value is higher than the index of the last register bank, then the highest possible register bank index is written. This 'saturated' value is the value read back from the STATUS32 register. For example, in a configuration with three register banks (banks 0, 1 and 2) and a STATUS32.RB update attempts to write the value 3: the write value is modified to 2 and register bank 2 is selected. The same mechanism is used when a debug host selects the register bank using the DEBUGI.RB field; any write of an index to a non-existent bank is saturated to the highest available bank. Note that an attempt to switch to a register bank that is not available is a programming error.
RBE	[19]	Register Bank Enable. When this bit is set, DEBUGI.RB value preempts STATUS32.RB for all instructions originating from the debug interface. When DEBUGI.RBE is not set, the existing value STATUS32.RB is used.
RESERVED	[31:20]	Reserved. Read-as-zero; ignored on write.

27

Debug Features

27.1 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

27.2 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 HS family processors, only the actionpoints that trigger on memory/auxiliary address and memory/auxiliary data values are post-commit; that is, they take effect after the triggering instruction completes. For more information about triggering delays, see the databook for your processor series.

27.2.1 Actionpoint Auxiliary Registers

Actionpoints are programmed using a collection of control and status registers mapped into the auxiliary register space.

Table 27-1 Auxiliary Registers for Actionpoints

Auxiliary Register Address	Register Name	Description
0x05	DEBUG	Extensions to Debug Register; (see Actionpoint Extensions to Debug Register, DEBUG, 0x05)
0x76	AP_BUILD	Actionpoint build configuration register(see Actionpoints Configuration Register, AP_BUILD)
0x220	AP_AMV0	Actionpoint 0 Match Value (see Actionpoint Match Value, AP_AMV0)
0x221	AP_AMM0	Actionpoint 0 Match Mask (see Actionpoint Match Mask, AP_AMM0)
0x222	AP_AC0	Actionpoint 0 Control (see Actionpoint Control, AP_AC0)
0x223	AP_AMV1	Actionpoint 1 Match Value (see Actionpoint Match Value, AP_AMV1)
0x224	AP_AMM1	Actionpoint 1 Match Mask (see Actionpoint Match Mask, AP_AMM1)
0x225	AP_AC1	Actionpoint 1 Control (see Actionpoint Control, AP_AC1)
0x226	AP_AMV2	Actionpoint 2 Match Value (see Actionpoint Match Value, AP_AMV2)
0x227	AP_AMM2	Actionpoint 2 Match Mask (see Actionpoint Match Mask, AP_AMM2)
0x228	AP_AC2	Actionpoint 2 Control (see Actionpoint Control, AP_AC2)
0x229	AP_AMV3	Actionpoint 3 Match Value (see Actionpoint Match Value, AP_AMV3)
0x22A	AP_AMM3	Actionpoint 3 Match Mask (see Actionpoint Match Mask, AP_AMM3)
0x22B	AP_AC3	Actionpoint 3 Control (see Actionpoint Control, AP_AC3)
0x22C	AP_AMV4	Actionpoint 4 Match Value (see Actionpoint Match Value, AP_AMV4)
0x22D	AP_AMM4	Actionpoint 4 Match Mask (see Actionpoint Match Mask, AP_AMM4)
0x22E	AP_AC4	Actionpoint 4 Control (see Actionpoint Control, AP_AC4)
0x22F	AP_AMV5	Actionpoint 5 Match Value (see Actionpoint Match Value, AP_AMV5)
0x230	AP_AMM5	Actionpoint 5 Match Mask (see Actionpoint Match Mask, AP_AMM5)
0x231	AP_AC5	Actionpoint 5 Control (see Actionpoint Control, AP_AC5)

Table 27-1 Auxiliary Registers for Actionpoints (Continued)

Auxiliary Register Address	Register Name	Description
0x232	AP_AMV6	Actionpoint 6 Match Value (see Actionpoint Match Value, AP_AMV6)
0x233	AP_AMM6	Actionpoint 6 Match Mask (see Actionpoint Match Mask, AP_AMM6)
0x234	AP_AC6	Actionpoint 6 Control (see Actionpoint Control, AP_AC6)
0x235	AP_AMV7	Actionpoint 7 Match Value (see Actionpoint Match Value, AP_AMV7)
0x236	AP_AMM7	Actionpoint 7 Match Mask (see Actionpoint Match Mask, AP_AMM7)
0x237	AP_AC7	Actionpoint 7 Control (see Actionpoint Control, AP_AC7)
0x23E	AP_ISO_EN	Actionpoint Isolation Enable (see Actionpoint Isolation Enable Register, AP_ISO_EN)
0x23F	AP_WP_PC	Watchpoint Program Counter (see Watchpoint Program Counter, AP_WP_PC)

**Note**

For more details on the access code definitions of the registers, see [Table 4-6 – “Key to Auxiliary Register Access Permissions for LR and SR Instructions”](#) on page 136.

27.2.1.1 Access Requirements

All actionpoint registers are accessible only in kernel mode; accessing these registers in user mode raises a Privilege Violation exception.

Vector Name	Vector Offset	Exception Cause Register
EV_PrivilegeV	0x1C	0x0700nn

The parameter field (nn) gives the number of the action point that triggered the exception.

- 0x00 = AP0
- 0x01 = AP1
- 0x02 = AP2
- 0x03 = AP3
- 0x04 = AP4
- 0x05 = AP5

- 0x06 = AP6
- 0x07 = AP7

If there are fewer than eight actionpoints, only the associated registers exist and accessing non-existent actionpoint registers raises an Illegal Instruction Error exception in both kernel and user modes.

Vector Name	Vector Offset	Exception Cause Register
Instruction Error	0x08	0x020000

27.2.1.2 Actionpoint Registers Host and Core Access

When the core is running, the enabled actionpoints must be reconfigured carefully because it may result in undefined behavior. The following is the recommended procedure:

- The host must first halt the core
- The host then checks the ASR status field and AH bit of the DEBUG register (see [Actionpoint Extensions to Debug Register, DEBUG, 0x05](#)) to see whether an actionpoint triggered during the halt, and takes appropriate action if necessary.

After these actions, the host can reconfigure actionpoints, and then restart the processor.

27.2.1.3 Actionpoint Registers Reset State

[Table 27-2](#) lists the values of the actionpoint registers when the ARCV2-based processor is reset.

Table 27-2 State of Auxiliary Registers upon Reset

Register Name	Initial Reset Values
DEBUG	0x00000000 for 2, 4 or 8 actionpoints
AP_AMV0 to AP_AMV7	0x00000000 for 2, 4 or 8 actionpoints
AP_AMM0 to AP_AMM7	0x00000000 for 2, 4 or 8 actionpoints
AP_AC0 to AP_AC7	0x00000000 for 2, 4 or 8 actionpoints
AP_BUILD	0x00000005 = 2 actionpoints, full targets 0x00000105 = 4 actionpoints, full targets 0x00000205 = 8 actionpoints, full targets 0x00000405 = 2 actionpoints, minimum targets 0x00000505 = 4 actionpoints, minimum targets 0x00000605 = 8 actionpoints, minimum targets

Table 27-2 State of Auxiliary Registers upon Reset (Continued)

Register Name	Initial Reset Values
AP_WP_PC	0x00000000

27.2.2 Actionpoint Extensions to Debug Register, DEBUG, 0x05

The debug register is standard in all ARCV2-based processors. The optional actionpoint system extends the [Debug Register, DEBUG](#) register (bits [10:2]) give additional status information for the actionpoint mechanism.

Each of the extension actionpoint fields in the DEBUG register performs the following functions:

- **AH:** The actionpoint halt flag (bit 2), of the DEBUG register, indicates that an actionpoint has halted the processor when the halt flag is set to 1. The AH flag is cleared when the halt flag (H) in the STATUS32 register (see [Status Register, STATUS32](#)) is cleared, for example, by restarting (see [Starting](#)) or single-stepping (see [Single Instruction Stepping](#)) the ARCV2-based processor. actionpoints that cause a software exception set the AH bit to 0.
- **ASR:** The actionpoints Status Register field (bits [10:3]) of the DEBUG register, indicates which actionpoints have triggered an actionpoint event, independent of whether the actionpoint raises a software exception or a processor halt. Each bit, from ASR0 to ASR7, indicates which actionpoints from 0 to 7 have triggered the actionpoint event. When actionpoints are linked forming pairs or quads, only the master actionpoint of the pair or quad has its ASR bit set when the actionpoint is triggered. The ASR bits are sticky, and Actionpoints trigger only when their corresponding ASR bit transitions from 0 to 1. Therefore, when an actionpoint condition is met and the ASR bit for the actionpoint is set to 1, the exception or breakpoint associated with that actionpoint is triggered only if ASR bit was previously 0. The ASR bit for an actionpoint is automatically cleared when the host or core writes to any of the actionpoint's associated control registers AP_AMVn, AP_AMMn, or AP_ACn (see [Actionpoint Match Value, AP_AMV0](#), [Actionpoint Match Mask, AP_AMM0](#), and [Actionpoint Control, AP_AC0](#)).

27.2.3 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 27-1 Actionpoint Build Register

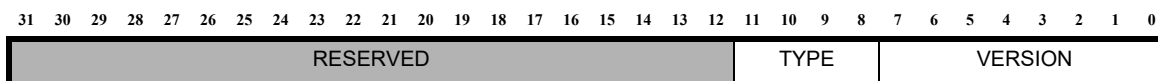


Table 27-3 Actionpoint Build Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Actionpoint Version 0x5: Version of the actionpoint system 0x7: Supports isolation of actionpoints
TYPE	[11:8]	Actionpoint Build Type Refers to the number of actionpoints selected in the ARCV2-based system: 0x0: 2 Actionpoints, full set of targets 0x1: 4 Actionpoints, full set of targets 0x2: 8 Actionpoints, full set of targets 0x4: 2 Actionpoints, minimum set of targets 0x5: 4 Actionpoints, minimum set of targets 0x6: 8 Actionpoints, minimum set of targets 0x3, 0x7 to 0xF: Reserved
RESERVED	[31:12]	Reserved, read as zero and ignored on writes

27.2.4 Actionpoint Match Value, AP_AMV0

Address: 0x220

Access: RW

Reset: 0x0000_0000

Figure 27-2 Actionpoint 0 Match Value Register



This register, in conjunction with the Actionpoint Match Mask register, specifies the condition that must be met for the associated actionpoint to trigger. This register contains the value that the monitored target must match.

After an actionpoint is triggered, the value in AP_AMV0 is modified to contain the value of the target parameter that caused the trigger. This is used by a debugger to determine the exact value that caused the trigger when the triggering condition specifies a range of addresses or data values.

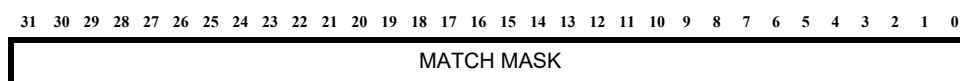
This register is set to 0x00000000 on reset.

You can read and write to this register.

27.2.5 Actionpoint Match Mask, AP_AMM0

Address: 0x221
 Access: RW
 Reset: 0x0000_0000

Figure 27-3 Actionpoint 0 Match Mask Register



This register contains the mask that is applied to both the monitored parameter and the match value (AP_AMV0) before the equality check is done.

At each bit position where a 1 is present, that bit plays *no role* in the comparison.

This register is set to 0x00000000 on reset.

This register is a read and write register.

For example, the following values specify a range of values from 0x12345000 to 0x12345FFF:

AP_AMV0 = 0x12345678

AP_AMM0 = 0x00000FFF

Depending on the setting of the Mode (M) bit in the Actionpoint Control register, the actionpoint triggers when the target value is within this range, or outside this range.

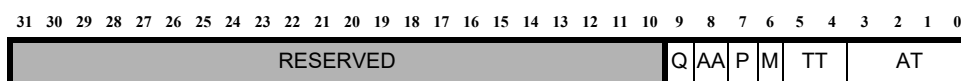
27.2.6 Actionpoint Control, AP_AC0

Address: 0x222

Access: RW

Reset: 0x0000_0000

Figure 27-4 Actionpoint 0 Control Register



Sets the target and operation mode.

This register is set to 0x00000000 on reset.

This register is a read and write register.

27.2.6.1 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 27-4](#) lists the selections available.

Table 27-4 Actionpoint Target Field, AT, Type

Actionpoint Type (AT)	Type of Data Comparison	Description
0x0	Instruction address	Instruction address trigger on specified PC value.
0x1	Instruction data	Instruction data trigger on specified instruction data.
0x2	Memory address	Memory address Trigger on access to specific memory address, from processor core only.
0x3	Memory data	Memory data trigger on a data value written to, or read from memory, from the processor core only. If both read and write are selected (TT=11) there is an implicit selection of write data only.
0x4	Auxiliary register address	Auxiliary register address trigger on access to specific auxiliary register address.
0x5	Auxiliary register data	Auxiliary register data trigger on data value written to, or read from an auxiliary register from the core only.
0x6	External parameter 0	External parameter 0 trigger on a specific value.
0x7	External parameter 1	External parameter 1 trigger on a specific value.
0x8 to 0xF	Reserved	N/A

**Caution**

When tracking accesses to the auxiliary registers (AT = 0100 or AT = 0101), only loads (LR) and stores (SR) to auxiliary registers can be monitored. Therefore, the STATUS32 register cannot be tracked when it is modified by the processor, that is, processor generated PC updates to the STATUS32 register.

27.2.6.2 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 27-5](#) lists the possible values: none (disabled), read, write, read or write.

Table 27-5 Transaction Type Field, TT, Type

Transaction Type (TT)	Description
0x0	Disable the associated actionpoint.
0x1	Trigger when the transaction is a write and the target value matches.
0x2	Trigger when the transaction is a read and the target value matches.
0x3	Trigger when the transaction is a write or read and the target value matches. If the write and read match conditions occur concurrently, the write values are implicitly selected for a match. If the write values do not match, the read values are not considered for a match.

27.2.6.3 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 27-6](#).

Table 27-6 Mode Field, M, Type

Mode (M)	Description
0x0	Trigger when in range
0x1	Trigger when outside range

27.2.6.4 Paired Field, P

The Pair (P) field (bit 7) of the AP_AC0 register, selects whether this actionpoint is a paired actionpoint. If an actionpoint is paired, both this actionpoint and the next one must meet their match conditions at the same time to cause a trigger (processor halt or software exception).

When actionpoint n (the *master* actionpoint in an actionpoint pair) is set to pair mode, it is combined with actionpoint $n+1$, to determine when an event occurs. If the P bit of the highest numbered actionpoint is set, it pairs with actionpoint 0. The possible pairings (for eight actionpoints) are therefore:

- (0,1)
- (1,2)
- (2,3)
- (3,4)
- (4,5)
- (5,6)
- (6,7)
- (7,0)

When the P bit for actionpoint n is set, the actionpoint $n+1$ is prevented from triggering on its own.

The P and Q bits of all actionpoints must be programmed such that there is no overlap in the actionpoint groups (pairs or quads) created. No actionpoint should belong to more than one such group. If such overlap is programmed, the triggering behavior of these groups is undefined.

When a paired actionpoint triggers, both actionpoint n and actionpoint $n+1$ have their respective AP_AMV registers updated with the appropriate target value.

The DEBUG ASR field and the ECR *nn* field only indicate the master of a pair that hits, that is, the actionpoint that has its P bit set. For example, if actionpoint 0 has its P bit set and an event occurs that matches the conditions of both actionpoint 0 and actionpoint 1, only bit 0 of the ASR is set.

A typical use of pairs is to detect writing of a particular value to a particular address or to more tightly specify a range of values than is possible to achieve with the Actionpoint Match Mask register of a single actionpoint.



Note

A pair containing one pre-commit actionpoint and one post-commit actionpoint never triggers because the former take effect before the triggering instruction completes; whereas the latter take effect after the triggering instruction completes. Therefore, the two triggering events never coincide.

Table 27-7 Pair Field, P, Type

Pair (P)	Description
0	Normal unpaired operation
1	Pair this actionpoint with the next actionpoint

27.2.6.5 Actionpoint Action Field, AA

The Actionpoint Action (AA) field (bit 8) of the AP_AC0 register defines whether the actionpoint causes the processor to halt, or raise an exception. Actionpoint halt in user mode requires that the `DEBUG.UB` bit is set.

Table 27-8 Actionpoint Action Field, AA Field, Type

Actionpoint Action (AA)	Description
0x0	Halt the processor If the processor is halted because of a Breakpoint, the instruction that caused the Breakpoint is not yet executed, and the PC continued to point at the triggering instruction. Watchpoints can halt the processor some cycles after the triggering instruction has been executed. In such case, the processor must be restarted from where it halted.
0x1	Raise an exception Cause a Privilege Violation exception (see Privilege Violation). This exception causes the program flow to be redirected through the EV_PrivilegeV vector. The Cause Code and Parameter field of the ECR indicate whether an Actionpoint was triggered and which Actionpoint caused the exception. A pre-commit Actionpoint exception is raised before the triggering instruction is executed, whereas a post-commit Actionpoint exception is raised after the triggering instruction has executed. Actionpoints in this mode are not triggered when the AE bit of the STATUS32 register is set. This means that they do not cause a double exception leading to a machine check.

27.2.6.6 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. Therefore, at least two of the four triggering events can never coincide.

Table 27-9 Quad Type

Quad (Q)	Description
0x0	Normal operation
0x1	Quad this actionpoint with the next three actionpoints (modulo the number of actionpoints)

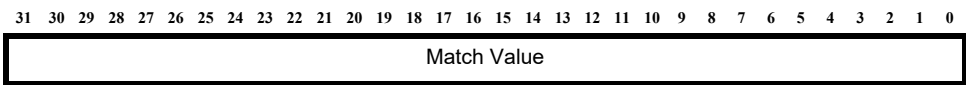
27.2.7 Actionpoint Match Value, AP_AMV1

Address: 0x223

Access: RW

Reset: 0x0000_0000

Figure 27-5 Actionpoint 1 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

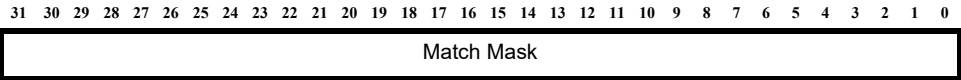
27.2.8 Actionpoint Match Mask, AP_AMM1

Address: 0x224

Access: RW

Reset: 0x0000_0000

Figure 27-6 Actionpoint 1 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

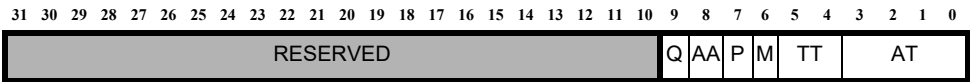
27.2.9 Actionpoint Control, AP_AC1

Address: 0x225

Access: RW

Reset: 0x0000_0000

Figure 27-7 Actionpoint 1 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

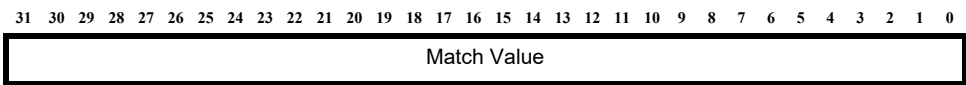
27.2.10 Actionpoint Match Value, AP_AMV2

Address: 0x226

Access: RW

Reset: 0x0000_0000

Figure 27-8 Actionpoint 2 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

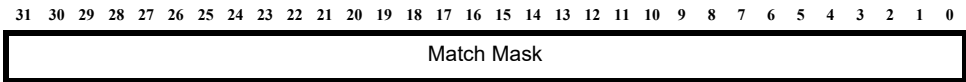
27.2.11 Actionpoint Match Mask, AP_AMM2

Address: 0x227

Access: RW

Reset: 0x0000_0000

Figure 27-9 Actionpoint 2 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

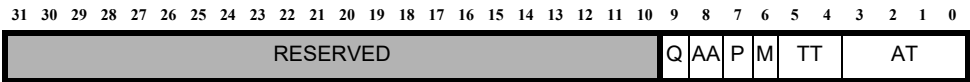
27.2.12 Actionpoint Control, AP_AC2

Address: 0x228

Access: RW

Reset: 0x0000_0000

Figure 27-10 Actionpoint 2 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

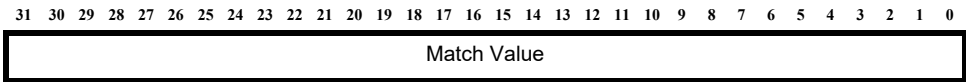
27.2.13 Actionpoint Match Value, AP_AMV3

Address: 0x229

Access: RW

Reset: 0x0000_0000

Figure 27-11 Actionpoint 3 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

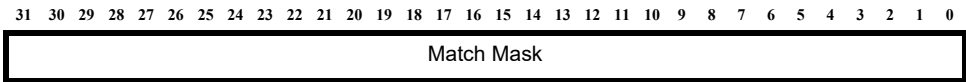
27.2.14 Actionpoint Match Mask, AP_AMM3

Address: 0x22A

Access: RW

Reset: 0x0000_0000

Figure 27-12 Actionpoint 3 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

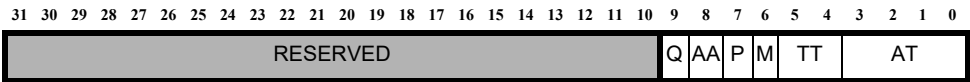
27.2.15 Actionpoint Control, AP_AC3

Address: 0x22B

Access: RW

Reset: 0x0000_0000

Figure 27-13 Actionpoint 3 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

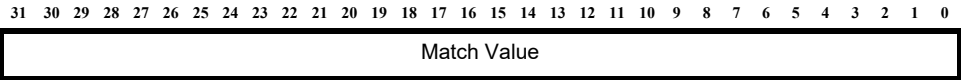
27.2.16 Actionpoint Match Value, AP_AMV4

Address: 0x22C

Access: RW

Reset: 0x0000_0000

Figure 27-14 Actionpoint 4 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

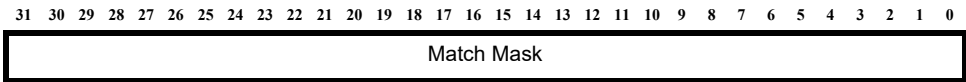
27.2.17 Actionpoint Match Mask, AP_AMM4

Address: 0x22D

Access: RW

Reset: 0x0000_0000

Figure 27-15 Actionpoint 4 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

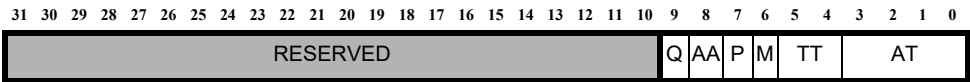
27.2.18 Actionpoint Control, AP_AC4

Address: 0x22E

Access: RW

Reset: 0x0000_0000

Figure 27-16 Actionpoint 4 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

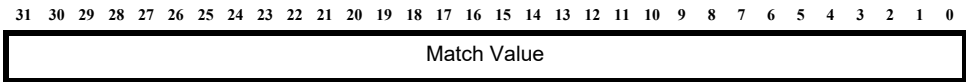
27.2.19 Actionpoint Match Value, AP_AMV5

Address: 0x22F

Access: RW

Reset: 0x0000_0000

Figure 27-17 Actionpoint 5 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

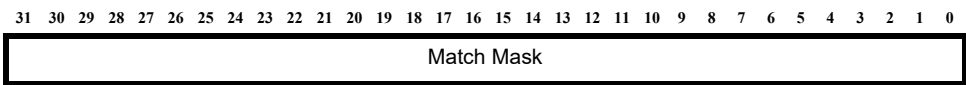
27.2.20 Actionpoint Match Mask, AP_AMM5

Address: 0x230

Access: RW

Reset: 0x0000_0000

Figure 27-18 Actionpoint 5 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

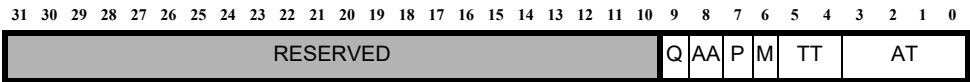
27.2.21 Actionpoint Control, AP_AC5

Address: 0x231

Access: RW

Reset: 0x0000_0000

Figure 27-19 Actionpoint 5 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

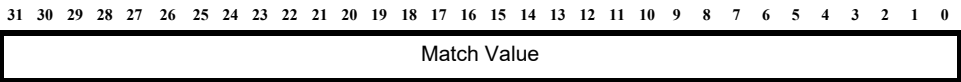
27.2.22 Actionpoint Match Value, AP_AMV6

Address: 0x232

Access: RW

Reset: 0x0000_0000

Figure 27-20 Actionpoint 6 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

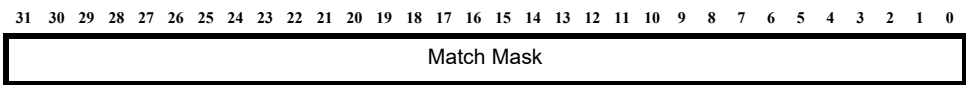
27.2.23 Actionpoint Match Mask, AP_AMM6

Address: 0x233

Access: RW

Reset: 0x0000_0000

Figure 27-21 Actionpoint 6 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

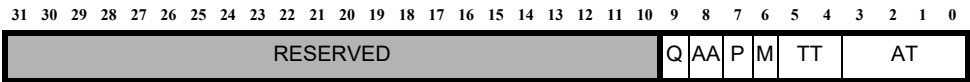
27.2.24 Actionpoint Control, AP_AC6

Address: 0x234

Access: RW

Reset: 0x0000_0000

Figure 27-22 Actionpoint 6 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

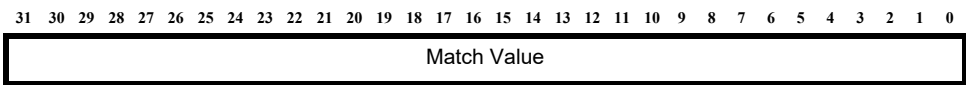
27.2.25 Actionpoint Match Value, AP_AMV7

Address: 0x235

Access: RW

Reset: 0x0000_0000

Figure 27-23 Actionpoint 7 Match Value Register



See [Actionpoint Match Value, AP_AMV0](#) register for field information.

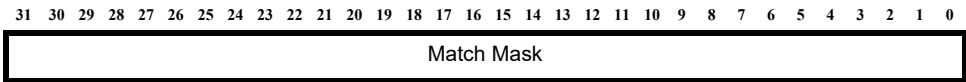
27.2.26 Actionpoint Match Mask, AP_AMM7

Address: 0x236

Access: RW

Reset: 0x0000_0000

Figure 27-24 Actionpoint 7 Match Mask Register



See [Actionpoint Match Mask, AP_AMM0](#) register for field information.

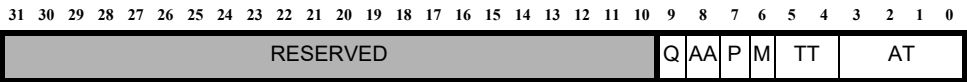
27.2.27 Actionpoint Control, AP_AC7

Address: 0x237

Access: RW

Reset: 0x0000_0000

Figure 27-25 Actionpoint 7 Control Register



See [Actionpoint Control, AP_AC0](#) register for field information.

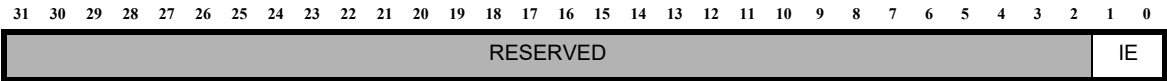
27.2.28 Actionpoint Isolation Enable Register, AP_ISO_EN

Address: 0x23E

Access: RW

Reset: 0x1

Figure 27-26 Actionpoint Isolation Enable Register



Use this register to enable or disable actionpoint isolation.



Note

Best practice is to disable actionpoints before enabling actionpoint isolation. If you have not done so, actionpoints might trigger two instructions before you disable actionpoint isolation. In such cases, add two NOP instructions before disabling actionpoint isolation.

Table 27-10 Actionpoint Isolation Enable Field Descriptions

Field	Bit	Reset	Description
IE	[1:0]	1	Isolation Enable (IE) field (bits 1:0) of the AP_ISO_EN register defines whether the isolation of action points is to be enabled or not. 0x0, 0x3: Invalid values; Programming these values generates parity error, which is aggregated in the safety monitor to generate <code>emsi_error</code> in builds where safety manager is present. Programming these values will be ignored, when safety manager is not present. 0x1: Disable isolation of actionpoints (default) 0x2: Enable isolation of actionpoints It is read as 0x1, when isolation is not enabled through this register field AND through the <code>secure_aps_iso_en</code> input signal. It is read as 0x2, when isolation is enabled either through this register field OR through the <code>secure_aps_iso_en</code> input signal. This register is not updated with 0x0 and 0x3
RESERVED	[31:2]	0	Reserved, read as zero and ignored on writes

27.2.29 Watchpoint Program Counter, AP_WP_PC

Address: 0x23F
 Access: R
 Reset: 0x0000_0000

Figure 27-27 AP_WP_PC, Watchpoint Program Counter Register when PC_SIZE = 32

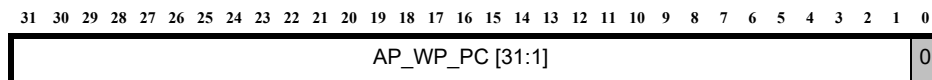
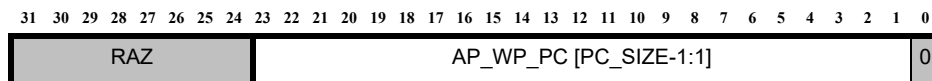


Figure 27-28 AP_WP_PC Register when PC_SIZE < 32



AP_WP_PC is a read-only register.

An ARCV2-based processor may execute one or two instructions after a watchpoint is triggered, because of the implementation-specific delay required to detect the watchpoint. It then either halts the processor or raises a Privilege Violation exception, depending on the Actionpoint Action (AA) bit of the triggered actionpoint.

To assist the debugger in identifying the location of the watchpoint-triggering instruction, the AP_WP_PC register contains the program address of the last instruction to trigger a watchpoint exception or halts.

Actionpoints on external parameters do not have an associated PC and for these actionpoints, the AP_WP_PC is undefined. Also, any graduating instruction data may complete out of order and a watchpoint triggered on such a instruction data does not have the instruction PC loaded in the AP_WP_PC register.

27.3 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.

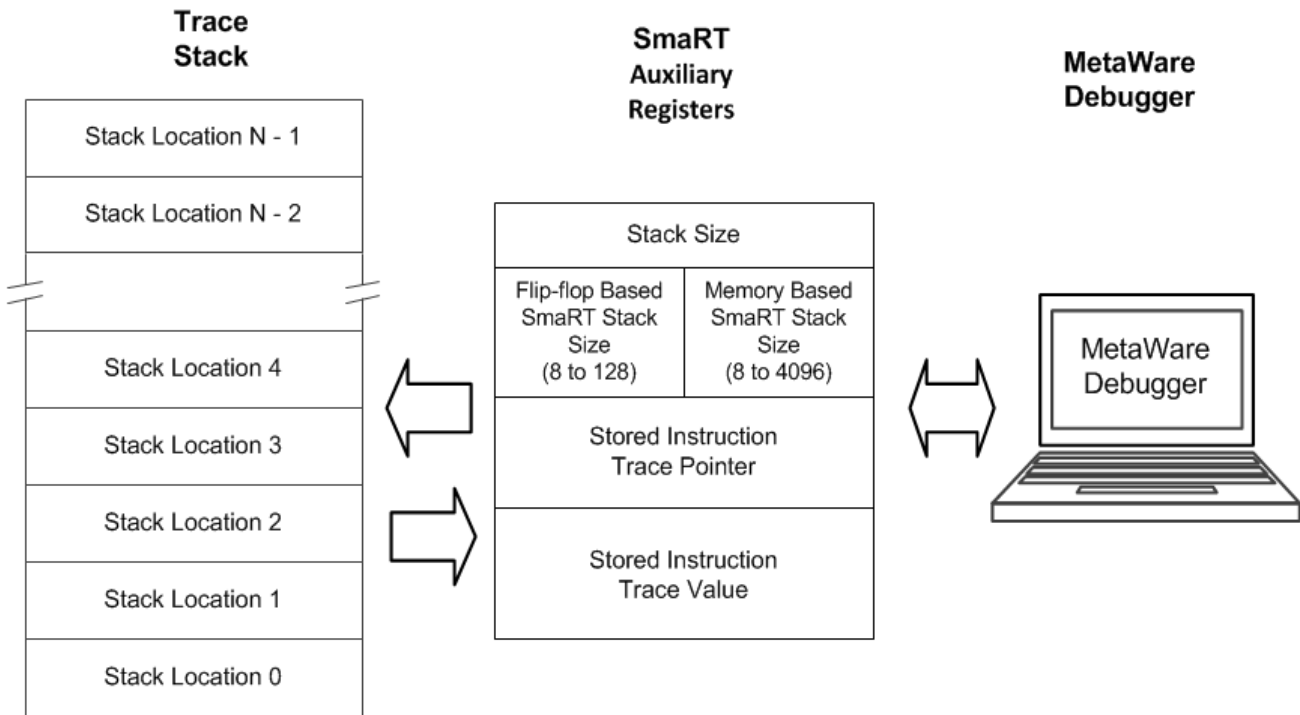
27.3.1 Features

- Stack size of 512 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.

27.3.2 Overview of SmaRT Operation

Figure 27-29 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

27.3.3 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

27.3.4 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 27-30 SMART_BUILD Value

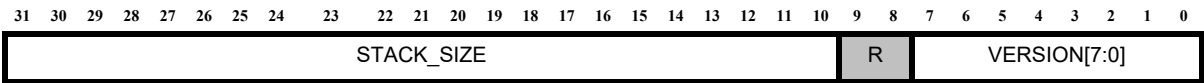


Table 27-11 lists the field descriptions:

Table 27-11 SMART_BUILD

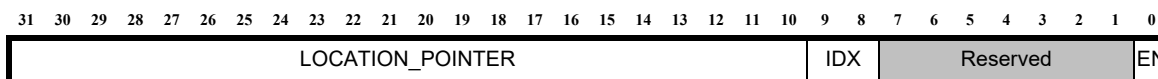
Field	Bits	Description
VERSION	[7:0]	0x3 initial version 0x4: SmArT now retains trace data even after the core is reset.
R	[9:8]	Reserved Returns zero when read.
STACK_SIZE	[31:10]	Trace stack size Indicates the stack size and allows the debugger to identify the capabilities of the implementation for display purposes. Possible values are from 8 up to 4096. Values above 128 are only available for the SRAM-based implementation.

27.3.5 SmaRT Control Register, SMART_CONTROL

Address: 0x700

Access: RW

Figure 27-31 SMART_CONTROL Register



This register lets you enable or disable SmaRT tracing.



Note

This register does not have a reset value. Hence, you must write to this register after reset before the first read attempt.

Table 27-12 lists the field descriptions:

Table 27-12 SMART_CONTROL Register

Field	Bits	Description
EN	[0]	SmaRT enable Enables the trace mechanism. ■ 0x0 : Trace Disabled. ■ 0x1 : Trace Enabled. Upon a hardware (core) reset, the EN bit is reset to 0 (smart trace function is disabled), but all entry data and pointers are retained to enable tracing the events that led to the hardware reset. Resetting smart trace data (entries) and internal entry read/write pointers can only be done using software reset: by toggling the EN bit from 0 to 1.
Reserved	[7:1]	Reserved Returns zero when reading. Ignored when writing.
IDX[1:0]	[9:8]	Location index Controls which value is returned in the SMART_DATA register. ■ 0x0: SRC_ADDR value ■ 0x1: DEST_ADDR value ■ 0x2: FLAGS value ■ 0x3: Reserved

Table 27-12 SMART_CONTROL Register

Field	Bits	Description
LOCATION_POINTER[21:0]	[31:10]	Stack location pointer Sets the value of the stored instruction stack pointer.



Note

Reading an entry outside the specified SmaRT stack size returns 0 data.

27.3.6 SmaRT Data Register, SMART_DATA

Address: 0x701

Access: R

The SMART_DATA read-only register has the following contents:

Figure 27-32 SMART_DATA Register

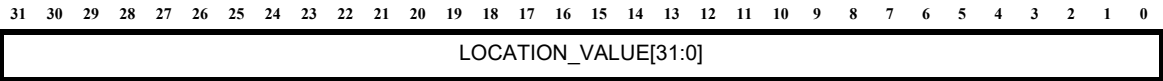


Table 27-13 lists the field descriptions:

Table 27-13 SMART_DATA Register

Field	Bits	Description
LOCATION_VALUE	[31:0]	Stored-instruction trace-stack value The value in the stack contained at the location given in the LOCATION_POINTER field and IDX field of the SMART_CONTROL register.

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.

27.3.7 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 27-33 SRC_ADDR Value

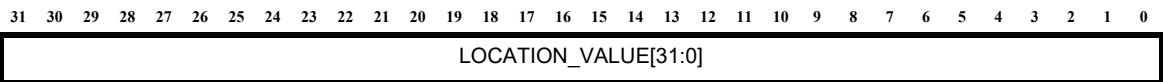


Table 27-14 lists the field descriptions:

Table 27-16 FLAGS Value Register

Field	Bits	Description
EX	[9]	Exception Indicates that the program-flow change is a result of an exception or interrupt.
RP	[10]	Repeat Used to indicate that the same program-flow change occurred more than once.
Reserved	[30:11]	Reserved Returns zero when read.
V	[31]	Valid Indicates that the value in the stack is valid.

27.3.9.1 Reserved Value, Index 3

Zero is read from the SMART_DATA register when IDX field in the SMART_CONTROL register is set to 3.

27.4 Debug Interface

When `dbg_en_option` this option is set to true, an external input signal is added to the BVCI interface. When set to high, debug access to the core through the ARC JTAG module is allowed. When cleared all debug accesses through the ARC JTAG module are silently ignored by the debug port and 0 is returned on the read data bus. The core also included an optional APB port using which you can debug the core. You can add the APB port using the `-dbg_apb_option==true`. The APB port has a select input (`dbg_prot_sel`) to control whether this new port or the existing BVCI port is active. Only one port can be active at any time.

The Debug interface is made available for systems that require access to the internals of the core. The interface is composed of memory mapped setup registers that can be written to initiate read and write operations to internal core/auxiliary registers and memory locations, as well as control operations such as instruction-stepping. The debug port is accessed from the JTAG port in the actual design, but may be alternatively connected to a simulation model for verification purposes only (using the `-fast_rascal` option).

This APB port allows access to the debug registers that initiate accesses to the core pipeline, as well as a CoreSight ROM table for device identification. It occupies a single 4 KB block within the Debug Access Port (DAP) address map. For multi-core builds, one 4 KB aperture is used for each core as each core has its own unique APB debug port.

All the internal cores and auxiliary registers of the ARC core, including those for programming any attached ARC Trace module, and any memories accessible from the core can be accessed through this interface. The memory map is the same as that used by the core.

27.4.1 ARC Debug-Access Registers

Table 27-17 on page 1169 lists the names, addresses, and reset values of the ARC debug-access registers.

Table 27-17 Debug Register Addresses

Auxiliary Address	Address over Debug Port	Reset value	Name
0xFFFF_0000	0x0000	0x24	DB_STATUS
0xFFFF_0004	0x0004	0x0	DB_CMD
0xFFFF_0008	0x0008	0x0	DB_ADDR
0xFFFF_000C	0x000C	0x0	DB_DATA
0xFFFF_0010	0x0010	0x0	DB_RESET

27.4.2 Debug Register, DB_STATUS

Address: 0xFFFF_0000

Access: RW

Reset: 0x0000_0000

The DB_STATUS register provides information on all DEBUG operations. Writing to this register initiates the debug operation defined in DB_CMD.

Figure 27-36 DB_STATUS Register

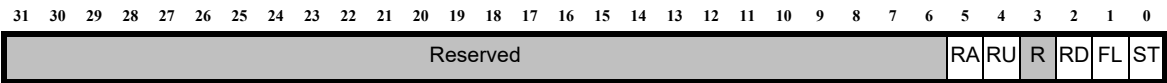


Table 27-18 DB_STATUS Field Description

Field	Bits	Reset	Description
ST	[0]	0	The stalled (ST) flag indicates that the debug interface is busy.
FL	[1]	0	When true, the failure (FL) flag indicates that the debug operation has failed
RD	[2]	0	The ready (RD) flag indicates whether the debug interface is available to accept another transaction command.
RESERVED	[3]	0	Reserved. Read-as-zero; ignored on write
RU	[4]	0	The run (RU) flag is set to one when the ARC HS processor is running
RA	[5]	0	The Reset Applied (RA) flag is used by the debug host to determine that a target reset has occurred.
RESERVED	[31:6]	0	Reserved. Read-as-zero; ignored on write

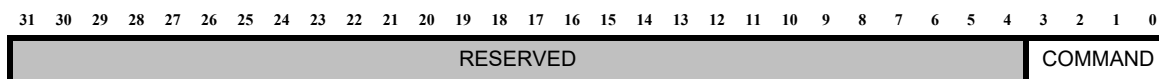
27.4.3 Debug Register, DB_CMD

Address: 0xFFFF_0004

Access: RW

Reset: 0x0000_0003

Figure 27-37 DB_CMD Register



The DB_CMD register defines the debug command (read/write to aux/core/mem).

Table 27-19 DB_CMD Field Description

Field	Bits	Reset	Description
COMMAND	[3:0]	0x3	Defines the debug command (read/write to aux/core/mem). All non-enumerated values result in a NOP instruction. ■ 0x0: Write to Memory Location ■ 0x1: Write to Core Register ■ 0x2: Write to Auxiliary Register ■ 0x4: Read Memory Location ■ 0x5: Read Core Register ■ 0x6: Read Auxiliary Register ■ 0x3, 0x7-0xF: Reserved
RESERVED	[31:4]	0	Reserved. Read-as-zero; ignored on write

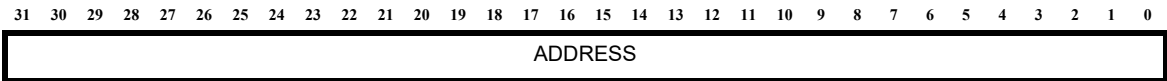
27.4.4 Debug Register, DB_ADDR

Address: 0xFFFF_0008

Access: RW

Reset: 0x0000_0000

Figure 27-38 DB_ADDR Register



The DB_ADDR register specifies auxiliary/core/memory address location.

Table 27-20 DB_ADDR Field Description

Field	Bits	Reset	Description
ADDRESS	[31:0]	0	Auxiliary/core/memory address location to be read from for debug read operations or written to for debug write operations.

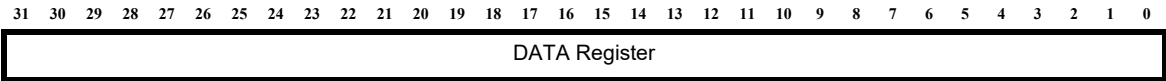
27.4.5 Debug Register, DB_DATA

Address: 0xFFFF_000C

Access: RW

Reset: 0x0000_0000

Figure 27-39 DB_DATA Register



DB_DATA specifies the data to be written for debug write operations. It also holds the returned data for debug read operations.

Table 27-21 DB_DATA Field Description

Field	Bits	Reset	Description
DATA_REGISTER	[31:0]	0	Data to be written for debug write operations or returned data for debug read operations.

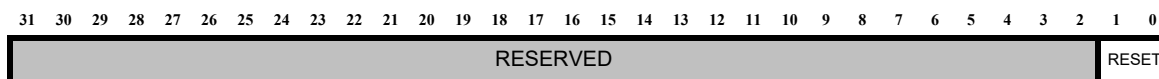
27.4.6 Debug Register, DB_RESET

Address: 0xFFFF_0010

Access: Write Only

Reset: 0x0000_0000

Figure 27-40 DB_RESET Register



Writing any non-zero value to the DB_RESET register triggers a system reset request.



Note

A BVCI write to this register resets the DB_CMD, DB_DATA and DB_ADDR registers.

Table 27-22 DB_ADDR Field Description

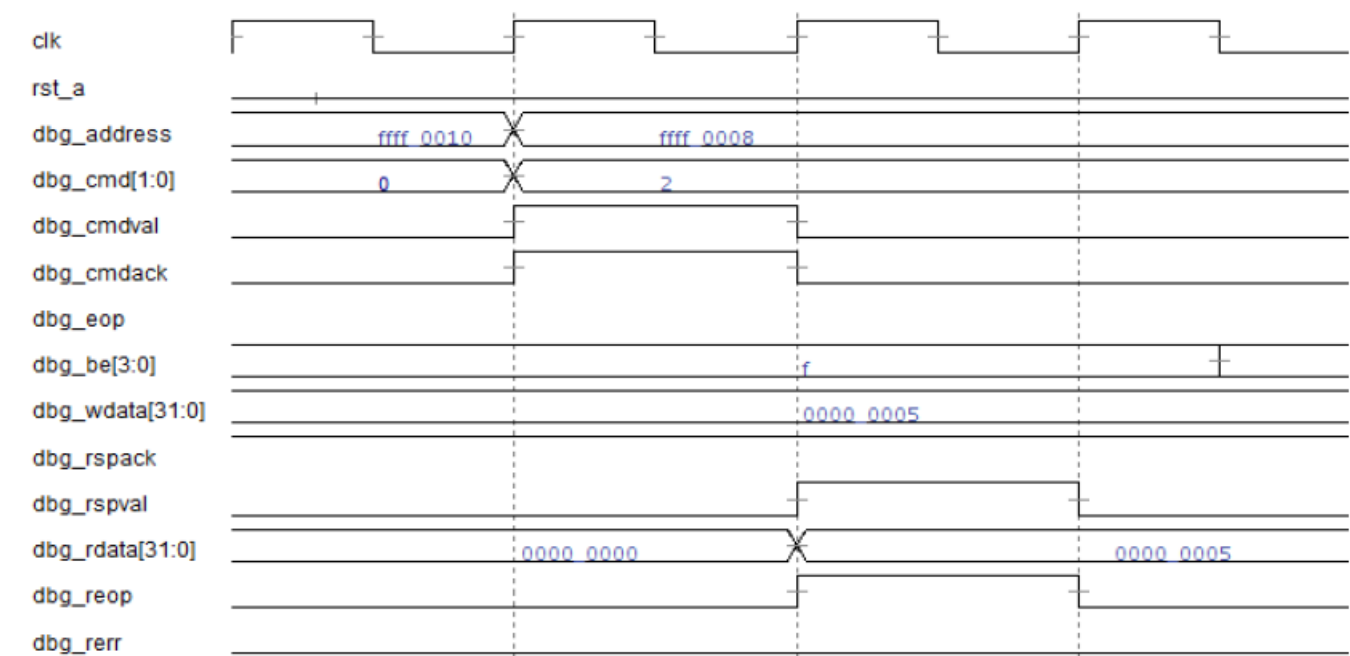
Field	Bits	Reset	Description
RESET	[1:0]	0	Triggers a system reset request. 0x0: No system reset request. 0x1: System reset request. 0x2: System reset request. 0x3: System reset request. All other values are reserved.
RESERVED	[31:2]	0	Reserved. Read-as-zero; ignored on write.

27.4.7 Initiating a Basic Debug Operation

A debug operation is set up by first writing to DB_ADDR, DB_DATA (for write operations), DB_CMD and finally DB_STATUS. For debug read operations, DB_DATA holds the data read from auxiliary, core register space or memory.

The busy or ready bit in DB_STATUS register should be read to determine whether the operation has completed. The fail bit should then be read to determine whether the operation succeeded or failed.

Figure 27-41 Debug Interface Timing Diagram



28

Performance Counters

28.1 Performance Counters Introduction

The ARCV2-based processors include a performance-monitoring system that enables rapid optimization. This system supports optimization of runtime software and can be used for the following:

- Hot-spot analysis — Locate the functions and instructions consuming the most CPU time. Hardware-assisted statistical code profiling provides visibility of instruction-execution frequencies. The system can also capture microarchitecture detail, reveal the average time taken for each instruction, and allow the root causes of instruction stalls and pipeline flushes to be identified.
- Instruction flow metrics — Identify patterns of behavior common to the whole system. An insight into the behavior of the system as a whole can be gained without requiring knowledge of any individual program block. System-level metrics are calculated from hardware counts of instruction and pipeline events. Example metrics include the following:
 - IPC (instructions per clock)
 - Cache-miss rates
 - Branch-prediction accuracy
 - Average load-use penalty
 - Breakdown of stall causes
 - Instruction mix (LD, ST, branch, and so on)
 - Instruction size (16 or 32 bits)
 - Average function size
 - Average instructions between branches
 - Average stack operations per call
 - Time spent in interrupt or exception handlers

Analysis of metrics can highlight techniques for improvement in the program code, compiler configuration, memory-subsystem design, and CPU configuration.

- **Real-time statistics display** — View system parameters live as the system runs. Metrics collected during a whole program run provide a picture of the system behavior. Graphs of metrics show change in real time as a response to workload or user interaction. Viewing plots of key metrics while the system runs lets you rapidly gain an impression of system behavior and response to workload, which can then be investigated further using more detailed analysis methods.

**Note**

You can control the dynamic power (execution control) consumption of the core by limiting the maximum instruction execution rate of the core. Use the [EXEC_CTRL](#) register to control the dynamic power consumption.

28.2 Performance Counters Registers

The following registers provided for interacting with the performance-counter block.

Table 28-1 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
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
0x25C	Interrupt Trigger Value, PCT_INT_CNTL	Interrupt trigger value low
0x25D	Interrupt Trigger Value, PCT_INT_CNTH	Interrupt trigger value high
0x25E	Interrupt Control Value, PCT_INT_CTRL	Interrupt control value
0x25F	Interrupt Active, PCT_INT_ACT	Interrupt active

Conditions Registers

The registers shown in [Table 28-2](#) are provided for interaction with the countable conditions block.

Table 28-2 Countable-Conditions Registers

Address	Name	Description
0xF6	Countable Conditions Build Configuration Register, CC_BUILD	Build-configuration register
0x240	Countable Conditions Index Register, CC_INDEX	Set index to access name of condition
0x241	Countable Conditions Name0 Register, CC_NAME0	Read first four characters of condition name, set by CC_INDEX
0x242	Countable Conditions Name1 Register, CC_NAME1	Read second four characters of condition name, set by CC_INDEX

28.2.1 Countable Conditions Index Register, CC_INDEX

Address: 0x240

Access: RW

Reset: 0x0

Figure 28-1 CC_INDEX Register



This register is used to set the index of the countable condition whose name is returned through the CC_NAME0 and CC_NAME1 registers. The conditions that are available is dependent on the processor configuration. For example, if caches are not available in the processor, the cache-related conditions are unavailable. The countable conditions do not have a fixed index. The mapping of named conditions to indexes can be discovered using the CC_INDEX and CC_NAME0 and CC_NAME1 registers.

Table 28-3 CC_INDEX Field Description

Field	Bit	Reset	Description
INDEX	[15:0]	0	Index of the countable condition whose name is returned through the registers CC_NAME0 and CC_NAME1. The index values start at zero.
RESERVED	[31:16]	0	Reserved. Read-as-zero; ignored on write.

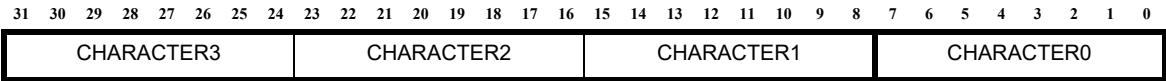
28.2.2 Countable Conditions Name0 Register, CC_NAME0

Address: 0x241

Access: R

Reset: 0x0

Figure 28-2 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 28-4 CC_NAME0 Field Description

Field	Bit	Reset	Description
CHARACTER0	[7:0]	0	Character 0
CHARACTER1	[15:8]	0	Character 1
CHARACTER2	[23:16]	0	Character 2
CHARACTER3	[31:24]	0	Character 3

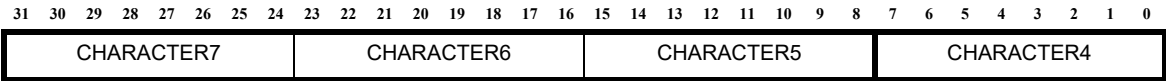
28.2.3 Countable Conditions Name1 Register, CC_NAME1

Address: 0x242

Access: R

Reset: 0x0

Figure 28-3 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 28-5 CC_NAME1 Field Description

Field	Bit	Reset	Description
CHARACTER4	[7:0]	0	Character 4
CHARACTER5	[15:8]	0	Character 5
CHARACTER6	[23:16]	0	Character 6
CHARACTER7	[31:24]	0	Character 7

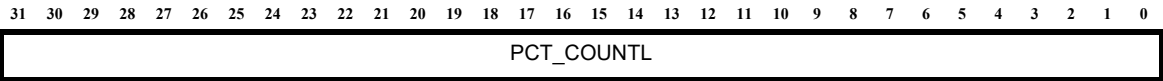
28.2.4 Count-Value Registers, PCT_COUNTL

Address: 0x250

Access: RW

Reset: 0x0

Figure 28-4 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 thePCT_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.

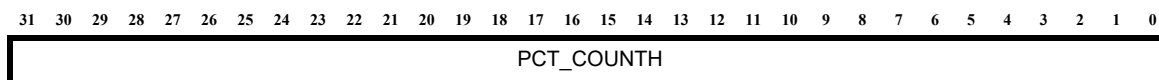
28.2.5 Count-Value Registers, PCT_COUNTH

Address: 0x251

Access: RW

Reset: 0x0

Figure 28-5 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.

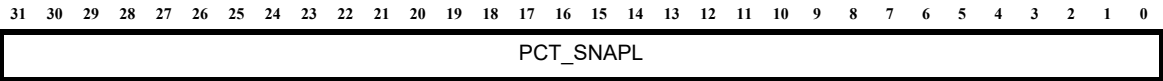
28.2.6 Snapshot-Value Registers, PCT_SNAPL

Address: 0x252

Access: RW

Reset: 0x0

Figure 28-6 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.

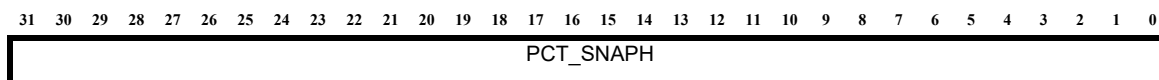
28.2.7 Snapshot-Value Registers, PCT_SNAPH

Address: 0x253

Access: RW

Reset: 0x0

Figure 28-7 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.

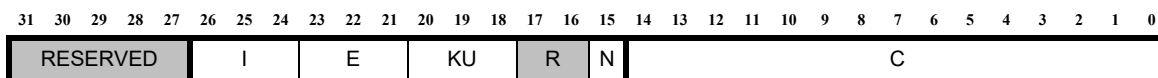
28.2.8 Configuration Register, PCT_CONFIG

Address: 0x254

Access: RW

Reset: 0x0

Figure 28-8 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 28-6 PCT_CONFIG Field Description

Field	Bit	Reset	Description
C	[14:0]	0	Countable condition index number
N	[15]	0	Invert condition before use
RESERVED	[17:16]	0	Reserved. Read-as-zero; ignored on write.
KU	[20:18]	0	0x0: count always 0x1: count only when in user mode 0x2: count only when in kernel mode 0x3 to 0x7: reserved
E	[23:21]	0	0x0: count always 0x1: count only when an exception is active 0x2: count only when an exception is not active 0x3 to 0x7: reserved
I	[26:24]	0	0x0: count always 0x1: count only when an interrupt is taken 0x2: count only when an interrupt is not taken 0x3 to 0x7: reserved
RESERVED	[31:27]	0	Reserved. Read-as-zero; ignored on write.

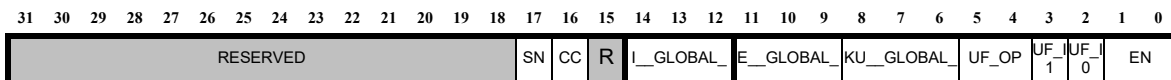
28.2.9 Control Register, PCT_CONTROL

Address: 0x255

Access: RW

Reset: 0x0

Figure 28-9 PCT_CONTROL Register



The global set counter run mode is used to choose all counters enable condition, such as count in kernel mode or user mode, count when an exception is active or inactive, or count when an interrupt is active or inactive. And for each counter, the PCT_CONFIG register controls the local counter enable conditions.

Note that features enabled/disable through programming this register would be effective after the write to this register is completed. For example - ISA events i4byte, i4lbyte, iall, isr, ilr might be triggered based on the instruction format that updates this register. If the instruction enables counting, the events generated by the instruction itself are not counted. If an instruction disables counting, the events are counted.

The counters only count when both local and global counter conditions can be met simultaneously. For example, if the local counter is programmed to count in the kernel mode and when in exception, and the global counter is programmed to count in kernel mode, then the counter counts only in the kernel mode and when in exception. If the global and local counters are programmed such that both the conditions cannot be met at the same time (for example, the local counter is set to count in user mode and the global counter is set to count in the kernel mode), the counter result is zero.

Table 28-7 PCT_CONTROL Field Description

Field	Bit	Reset	Description
EN	[1:0]	0	Enable condition for all counters 0x0: Disable counting 0x1: Enable counting (global enable) 0x2: Count when uflag 0 (user flag 0) and uflag 1 (user flag 1) conditions (as defined in bits [5:2] of this register) are met. 0x3: Global set count run mode (U/K/1, E/NE/1, I/NI/1) This bit has read and write access.
UF_I0	[2]	0	0x0: Use uflag0 as is in the condition UF_OP (bits [5:4]) 0x1: Invert uflag0 (UF_I0) before using in the condition UF_OP (bits [5:4])

Table 28-7 PCT_CONTROL Field Description

Field	Bit	Reset	Description
UF_I1	[3]	0	0x0: Use uflag1 as is in the condition UF_OP (bits [5:4]) 0x1: Invert uflag1 (UF_I1) before using in the condition UF_OP (bits [5:4])
UF_OP	[5:4]	0	User flag operand 0x0: enable counting when (uflag0 XOR UF_I0) AND (uflag1 XOR UF_I1) == 1 0x1: enable counting when (uflag0 XOR UF_I0) OR (uflag1 XOR UF_I1) == 1 0x2: enable counting when (uflag0 XOR UF_I0) == 1 0x3: enable counting when (uflag1 XOR UF_I1) == 1
KU_GLOBAL_	[8:6]	0	0x0: count always 0x1: count only when in user mode 0x2: count only when in kernel mode 0x3-0x7: reserved
E_GLOBAL_	[11:9]	0	0x0: count always 0x1: count only when an exception is active 0x2: count only when an exception is NOT active 0x3-0x7: reserved
I_GLOBAL_	[14:12]	0	0x0: count always 0x1: count only when an interrupt is active 0x2: count only when an interrupt is NOT active 0x3-0x7: reserved
RESERVED	[15]	0	Reserved. Read-as-zero; ignored on write.
CC	[16]	0	Clear counts. When this bit is written with 1, all counters are cleared to zero. Write-only – returns zero on read
SN	[17]	0	Snapshot. When this bit is written with 1, all count values are copied into the snapshot registers. Write-only – returns zero on read
RESERVED	[31:18]	0	Reserved. Read-as-zero; ignored on write.

28.2.10 Index-Select Register, PCT_INDEX

Address: 0x256

Access: RW

Reset: 0x0

Figure 28-10 PCT_INDEX Register



This register is used to select the counters to be accessed through count, snapshot, and configuration registers.

Table 28-8 PCT_INDEX Field Description

Field	Bit	Reset	Description
B	[15:0]	0	Counter number. Selects the counters to be accessed through count, snapshot, and configuration registers. Counter operation is not affected by index selection.
RESERVED	[31:16]	0	Reserved. Read-as-zero; ignored on write.

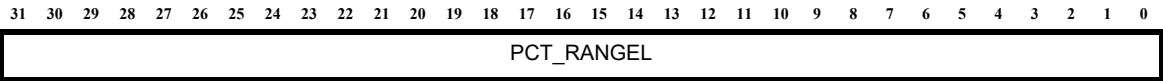
28.2.11 Address-Range Registers, PCT_RANGE[0-31]

Address: 0x259

Access: RW

Reset: 0x0

Figure 28-11 PCT_RANGE[0] 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 $(PCT\_RANGE[0] \leq PC)$ and either $(PC < PCT\_RANGEH)$ or $(PCT\_RANGEH == 0)$.

This register defaults to 0x0 on reset.

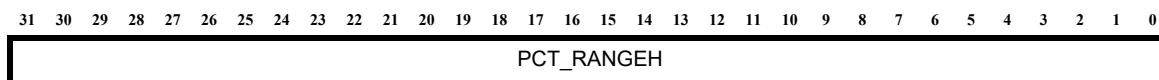
28.2.12 Address-Range Registers, PCT_RANGEH

Address: 0x25A

Access: RW

Reset: 0x0

Figure 28-12 PCT_RANGEH Register



Address range registers can be used to enable counting only when the program counter is within a specific address range. Counting is enabled when $(PCT\_RANGEH \leq PC)$ and either $(PC < PCT\_RANGEH)$ or $(PCT\_RANGEH == 0)$.

Note that PCT_RANGEH is the first instruction outside the range, so to set the high end of the range to the top of memory, set it to 0 (as the first instruction after the top of memory is $0xffffffff+2=0x0$).

This register defaults to 0x0 on reset.

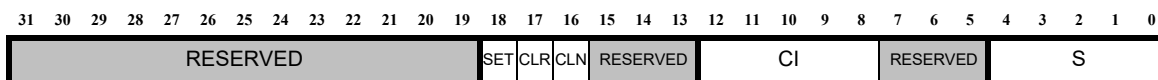
28.2.13 User-Flag Register, PCT_UFLAGS

Address: 0x25B

Access: RW

Reset: 0x0

Figure 28-13 PCT_UFLAGS Register



User-settable flags (uflag x) are provided for instrumenting a specific program. [Table 28-9](#) shows the bit positions for writes.

When the register is read, the current state of the 32-bit flag vector is returned.



Note

You can use the uflag0 and uflag1 (by programming the PCT_CONTROL register) to control the counting of conditions.

The register allows a single aux write to perform both set and clear operations simultaneously. [Example 28-1](#) describes a few example.

Example 28-1 Example Operations

One-hot operation. Clear all bits, then set flag number N :

```
_sr( (1<<16) | N , AUX_PCT_UFLAGS)
```

Single-bit operation. Clear bit N :

```
_sr( (1<<17) | (N<<8) , AUX_PCT_UFLAGS)
```

Single-bit operation. Set bit N :

```
_sr( (1<<18) | N , AUX_PCT_UFLAGS)
```

Multi-bit operation, Clear bit M , Set bit N :

```
_sr(1<<17) | (M<<8) | (1<<18) | N , AUX_PCT_UFLAGS)
```

Uses

User flags (uflag x) can be used in the following ways:

- Performance counters and ARCPlotter
 - Instrument code into different sections by setting flags, such as uflag0=encode, uflag1=decode, uflag2=protocol, and so on.
 - Counters are programmed to count cycles when uflag0/1 are set.

- Results can be reported at the end of the run, or displayed in real time using the MetaWare debugger's ARCPlotter feature.

**Note**

Only the MetaWare debugger provides live, runtime plotting.

- Performance counters.
 - Instrument code into different sections by setting flags.
 - The counters can be enabled by a condition (or the inverse of a condition).
 - You can then rapidly enable counting for different sections of code by programming the counter hardware to capture data only during `uflag0/uflag1/ . .` time periods. Different sections can be selected without changing the code.

Table 28-9 PCT_UFLAGS Field Description

Field	Bit	Reset	Description
S	[4:0]	0	Set-flag index
RESERVED	[7:5]	0	Reserved. Read-as-zero; ignored on write.
CI	[12:8]	0	Clear-flag index
RESERVED	[15:13]	0	Reserved. Read-as-zero; ignored on write.
CLN	[16]	0	Clear all flags
CLR	[17]	0	Clear flag specified by CI field
SET	[18]	0	Set flag specified by the S field, after clearing operations have taken place
RESERVED	[31:19]	0	Reserved. Read-as-zero; ignored on write.

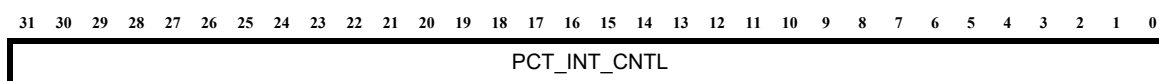
28.2.14 Interrupt Trigger Value, PCT_INT_CNTL

Address: 0x25C

Access: RW

Reset: 0x0

Figure 28-14 PCT_INT_CNTL Register



Performance counters can be configured to trigger interrupt when a count is reached. Each count value is an unsigned integer, presented in two 32-bit registers. The counter to be accessed is controlled by the PCT_INDEX register:

- The lower-order bits are provided in register PCT_INT_CNTL.
- The higher-order bits are provided in register PCT_INT_CNTH.

Count values wrap to zero when the maximum count value is reached.

The enable bit corresponding to the counter should also be set for the interrupt to be triggered.

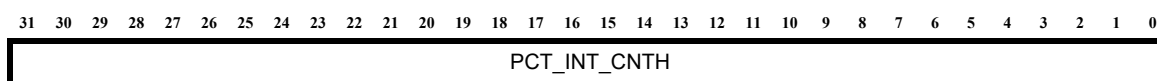
28.2.15 Interrupt Trigger Value, PCT_INT_CNTH

Address: 0x25D

Access: RW

Reset: 0x0

Figure 28-15 PCT_INT_CNTH Register



Performance counters can be configured to trigger interrupt when a count is reached. Each count value is an unsigned integer, presented in two 32-bit registers. The counter to be accessed is controlled by the PCT_INDEX register:

- The lower-order bits are provided in register PCT_INT_CNTL.
- The higher-order bits are provided in register PCT_INT_CNTH.

Count values wrap to zero when the maximum count value is reached.

The enable bit corresponding to the counter should also be set for the interrupt to be triggered.

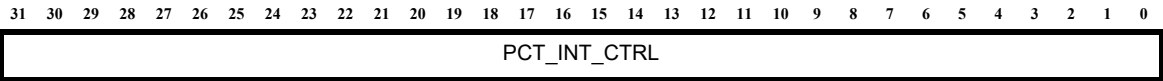
28.2.16 Interrupt Control Value, PCT_INT_CTRL

Address: 0x25E

Access: RW

Reset: 0x0

Figure 28-16 PCT_INT_CTRL Register



Each bit in this register is an interrupt enable bit for performance counter *X*, where *X* is the counter number.

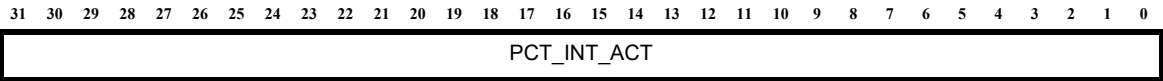
28.2.17 Interrupt Active, PCT_INT_ACT

Address: 0x25F

Access: RW

Reset: 0x0

Figure 28-17 PCT_INT_ACT Register



This register indicates the interrupts that are active (triggered).

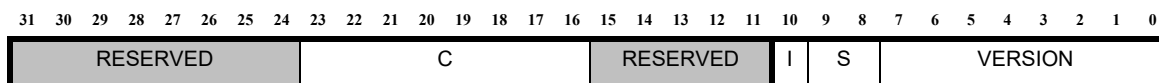
Field	Bit	Reset	Description
PCT_INT_ACT	[31:0]	0	Indicates the counters that have triggered an interrupt. For example, if counter <i>X</i> has triggered an interrupt, the PCT_INT_ACT[<i>X</i>] bit is set to 1. Software must write a 1 to the appropriate bit in the PCT_INT_ACT register to clear the corresponding interrupt line. If multiple counters trigger the PCT interrupt, then all active bits should be cleared to clear the PCT Interrupt. Clearing an interrupt through PCT_INT_ACT also disables the corresponding interrupt in PCT_INT_CTRL.

28.2.18 Performance Counter Build-Configuration Register, PCT_BUILD

Address: 0xF5

Access: R

Figure 28-18 PCT_BUILD Register



The performance-counter build-configuration register contains the fields shown in [Table 28-10](#) on page 1199.

Table 28-10 PCT_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number. All non-enumerated values are reserved. 0x0: not present 0x3: Current version of the performance counter component
S	[9:8]	Counter size: All non-enumerated values are reserved. 0x0: 32 bit 0x1: 48 bit (default) 0x2: 64 bit All other values – reserved
I	[10]	Indicates the ability of the PCT logic to generate an interrupt on counter overflow. 0x0: interrupt capability excluded 0x1: interrupt capability included
RESERVED	[15:11]	Reserved. Read-as-zero; ignored on write.
C	[23:16]	Number of performance counters. All non-enumerated values are reserved. 0x0: 0 performance counters 0x2: 2 performance counters 0x4: 4 performance counters 0x8: 8 performance counters 0x10: 16 performance counters 0x20: 32 performance counters
RESERVED	[31:24]	Reserved. Read-as-zero; ignored on write.

28.2.19 Countable Conditions Build Configuration Register, CC_BUILD

Address: 0xF6

Access: R

Figure 28-19 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 28-11 CC_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number. All non-enumerated values are reserved. ■ 0x0 if no countable conditions are present ■ 0x2 for countable conditions
RESERVED	[15:8]	Reserved. Read-as-zero; ignored on write.
CC	[31:16]	The number of countable conditions present in the system

29

Processor Timers

The processor timers are two independent 32-bit timers and a 64-bit real-time counter (RTC). Timer 0 and timer 1 are identical in operation. The only difference is that these timers are connected to different interrupts. The Timers cannot be included in a configuration without interrupts. Each timer is optional and when present, it is connected to a fixed interrupt; interrupt 16 for timer 0 and interrupt 17 for timer 1.

29.1 Timers

The processor timers are connected to a system clock signal that operates even when the ARCV2-based processor is in some (but not all) of the processor sleep states. The watchdog timer, if configured, can be connected to the same system clock as the other timers, or an external clock signal. In the sleep states where the timers are enabled, the timers can be used to generate interrupt signals that wake the processor from the SLEEP state. For more information about internal clocks to the timers during the sleep state, see [Table 26-3](#) on page 1121.



Note

When the processor is in the HALT state, timers do not count up or down, but the registers can be accessed by the debugger. This means that the timer interrupts (including a watchdog interrupt, if the watchdog is configured to be internally clocked), are not generated when the processor is halted.

The processor timers automatically reset and restart their operation after reaching the limit value. The processor timers can be programmed to count only the clock cycles when the processor is not halted. The processor timers can also be programmed to generate an interrupt or to generate a system [Reset](#) upon reaching the limit value.

The 64-bit RTC does not generate any interrupts. This timer is used to count the clock cycles atomically.

29.1.0.1 Programming

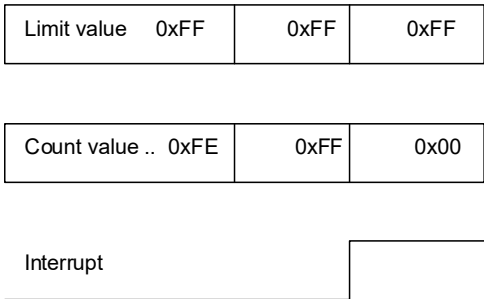
To program a timer n , use the following sequence:

- 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. COUNT n can be set greater than LIMIT n . In this case, Timer n starts incrementing from COUNT n before rolling over the upper limit of the register until it reaches LIMIT n .

Figure 29-1 Interrupt Generated after Timer Reaches Limit Value



The software must clear the timer interrupts by clearing CONTROL n .IP bit. After an interrupt is generated, write to the CONTROL n register to clear the timer count. This action must be performed during the interrupt service routine.

29.2 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 29-1 and Table 29-2.

Table 29-1 Auxiliary Registers for Hardware Counter 0 (has_timer_0 == true)

Address	Auxiliary Register Name	Description
0x21	Timer 0 Count Register, COUNT0	Processor timer 0 count value
0x22	Timer 0 Control Register, CONTROL0	Processor timer 0 control value
0x23	Timer 0 Limit Register, LIMIT0	Processor timer 0 limit value

Table 29-2 Auxiliary Registers for Hardware Counter 1 (has_timer_1 == true)

Address	Auxiliary Register Name	Description
0x100	Timer 1 Count Register, COUNT1	Processor timer 1 count value
0x101	Timer 1 Control Register, CONTROL1	Processor timer 1 control value
0x102	Timer 1 Limit Register, LIMIT1	Processor timer 1 limit value

Table 29-3 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

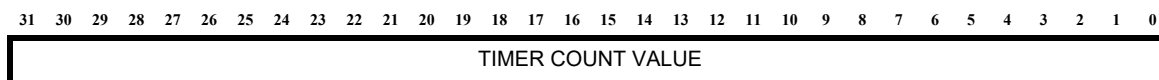
29.2.1 Timer 0 Count Register, COUNT0

Address: 0x21

Access: RW

Reset: 0x0000_0000

Figure 29-2 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.

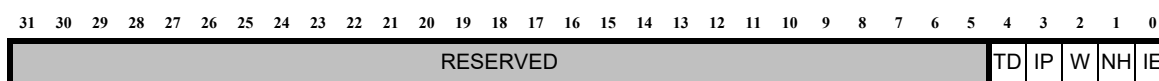
29.2.2 Timer 0 Control Register, CONTROL0

Address: 0x22

Access: RW

Reset: 0x0000_0000

Figure 29-3 Timer 0 Control Register



The timer control register (CONTROL0) is used to update the control modes of the timer.

Writing to CONTROL0 de-asserts the timer interrupt, but does not stop the timer from counting. The timer continues counting and independently starts the next iteration of counting, setting COUNT0 to 0, when LIMIT0 equals COUNT0.

If both the IE and W bits are set, only the watchdog reset is activated because the ARCV2-based processor has been reset and the interrupt is lost. Regardless of the IE and W bits, the timer is automatically reset and the timer restarts its operation after reaching the limit value.

All of the control flags must be programmed in one write access to the CONTROL0 register.

Table 29-4 TIMER 0 Field Descriptions

Field	Bit	Reset	Description
IE	[0]	0	The Interrupt Enable flag (IE) enables the generation of an interrupt after the timer has reached its limit condition. If this bit is not set, no interrupt is generated. When the processor is Reset , the IE flag is set to 0.
NH	[1]	0	The Not Halted mode flag (NH) causes cycles to be counted only when the processor is running (that is when the processor is not halted). When set to 0, the timer counts every clock cycle. When set to 1, the timer counts only when the processor is running. If the NH flag is set to 1, counting is suspended during host debugger interactions with the processor. However, if the NH flag is set to 0, counters are free-running. When the processor is Reset , the NH flag is set to 0.
W	[2]	0	The Watchdog mode flag (W) enables the generation of a system watchdog reset signal after the timer has reached its limit condition. If this bit is not set, no watchdog reset signal is generated. The watchdog reset signal is activated two cycles after the limit condition is reached. The watchdog reset signal can be used to cause a system or processor Reset with appropriate external logic.

Table 29-4 TIMER 0 Field Descriptions (Continued)

Field	Bit	Reset	Description
IP	[3]	0	The IP bit is set when the COUNTn register reaches the LIMITn value, and remains set until cleared by the timer interrupt service routine. The IP bit can be cleared by re-writing the desired values of W, NH and IE into the CONTROL0 register, thereby writing a 0 into the IP position. An interrupt service routine typically clears the IP bit by reading the CONTROL0 register, masking out the IP bit, and writing the resulting value back to the CONTROL0 register.
TD	[4]	0	The TD bit controls the behavior of the timer when the core is in a power-down mode. When set to 0, the timer continues to count when the core is in the power-down mode. When set to 1, the timer clock is disabled and gated to save power and the counting is discontinued. When the processor is in sleep mode 0 or sleep mode 2, the timer is enabled explicitly irrespective of the TD bit value. The TD bit affects the Level 1 clock gating function for the CPU Core (also controlled by “ Global Clock Disable Register, GLOBAL_CLK_GATE_DIS ” register). The default value (0) of the TD bit defeats the Level 1 clock gating function of the core such that the Level 1 (main) clock to the core, remains on during core idle periods (halt, sleep). In this case, the timer continues to count during idle periods. When the TD bit is enabled, the Level 1 core clock is allowed to turn off during idle periods as controlled by “ Global Clock Disable Register, GLOBAL_CLK_GATE_DIS ” register and the idle state of the core.
RESERVED	[31:4]	0	Reserved. Read-as-zero; ignored on write.

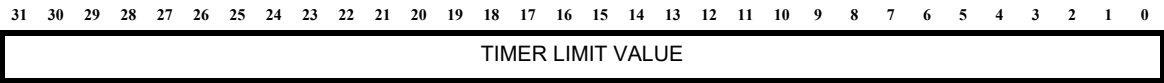
29.2.3 Timer 0 Limit Register, LIMIT0

Address: 0x23

Access: RW

Reset: 0x00FFFFFF

Figure 29-4 Timer 0 Limit Value Register

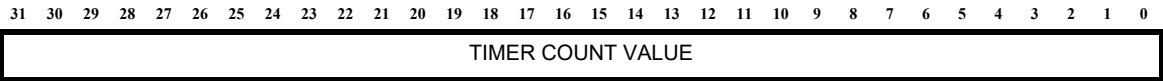


You must write the limit value into this register. The limit value is the value after which an interrupt or a reset must be generated. For backward compatibility to previous processor variants, the timer limit register is set to 0x00FFFFFF when the processor is [Reset](#).

29.2.4 Timer 1 Count Register, COUNT1

Address: 0x100
Access: RW
Reset: 0x0000_0000

Figure 29-5 Timer 1 Count Value Register



See [Timer 0 Count Register, COUNT0](#) on [page 1204](#) for field information.

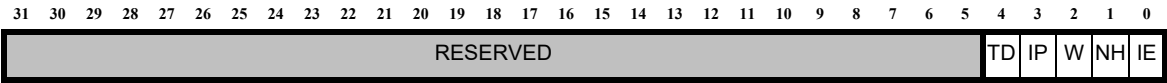
29.2.5 Timer 1 Control Register, CONTROL1

Address: 0x101

Access: RW

Reset: 0x0000_0000

Figure 29-6 Timer 0 Control Register



See [Timer 0 Control Register, CONTROL0](#) on [page 1205](#) for field information.

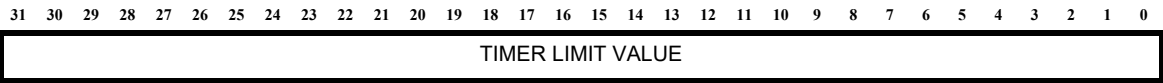
29.2.6 Timer 1 Limit Register, LIMIT1

Address: 0x102

Access: RW

Reset: 0x00FFFFFF

Figure 29-7 Timer 1 Limit Value Register



See [Timer 0 Limit Register, LIMIT0](#) on [page 1207](#) for field information.

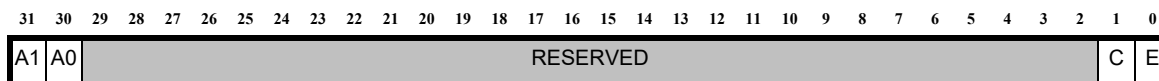
29.2.7 RTC Control Register, AUX_RTC_CTRL

Address: 0x103

Access: rW

Reset: 0x0000_0000

Figure 29-8 AUX_RTC_CTRL Register

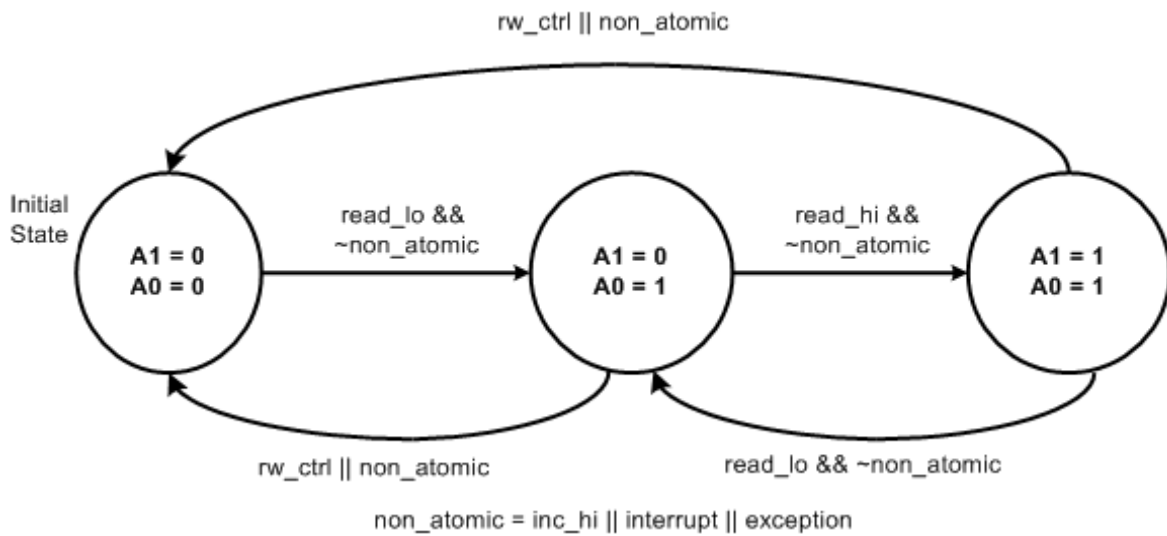


The 64-bit RTC is a free-running counter. This counter does not have any associated interrupts. The RTC is controlled using the AUX_RTC_CTRL register. The following are the field descriptions:

Table 29-5 RTC Control Register

Fields	Bit	Reset	Description
E	[0]	0	Enable A value of 0 means disabled; 1 means enable counting.
C	[1]	0	A value of 1 clears the AUX_RTC_LOW and AUX_RTC_HIGH registers.
RESERVED	[29:2]	0	Reserved. Read-as-zero; ignored on write.
A1, A0	[31:30]	0	These bits track the atomicity of reads from the RTC Count High Register, AUX_RTC_HIGH and RTC Count Low Register, AUX_RTC_LOW registers. For the description of the state machine, see Figure 29-9 on page 1212 .

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 29-9 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).

29.2.7.1 Reading RTC Atomically

The following example code explains how to read the RTC atomically:

```
RTC_read: LR R0, [AUX_RTC_LOW]
          LR R1, [AUX_RTC_HIGH]
          LR R2, [AUX_RTC_CTRL]
          BBIT0.NT R2, 31, RTC_read
```

- If an interrupt occurs between the first two LR instructions, R2 does not have bit 31 set to 1, and the BRLT loops back to enforce a re-read.
- If an exception occurs when executing any of the LR instructions, a loop back is enforced to re-read.
- If AUX_RTC_HIGH is incremented after reading AUX_RTC_LOW, again a re-read is enforced.
- Due to the infrequency of interrupts and increments to AUX_RTC_HIGH, this loop rarely requires more than 1 iteration.
- As the branch is rarely taken, a static prediction of .NT must be given. This implementation improves the performance on implementations with deeper pipelines.

**Note**

Placing the reads inside a critical region ensures atomicity, but does not detect when the AUX_RTC_LOW register wraps around between the two reads. Therefore, always use the above scheme.

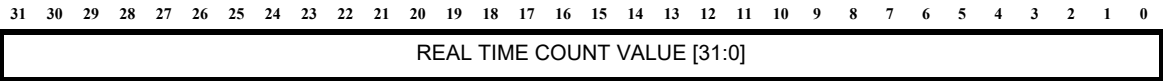
29.2.8 **RTC Count Low Register, AUX_RTC_LOW**

Address: 0x104

Access: r

Reset: 0x0000_0000

Figure 29-10 RTC Count Low Register

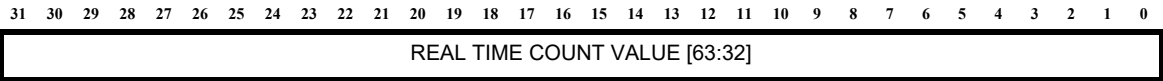


The RTC count low register is a read register in user mode. Reading this register returns the LSB 32-bits of the free-running RTC.

29.2.9 RTC Count High Register, AUX_RTC_HIGH

Address: 0x105
Access: r
Reset: 0x0000_0000

Figure 29-11 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.

29.2.10 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 29-12 TIMER_BUILD Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED								P1			P0			RESERVED				RTC	T1	T0	VERSION										

Table 29-6 TIMER_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Current version. All non-enumerated values are reserved. 0x1: Version 1 0x2: ARC 700 Processor Timers 0x3: ARC 600 R3 Processor Timers, with interrupt-pending bits 0x4: ARCV2 Processor Timers 0x6: ARCV2 Timer control register with the TD bit implementation to explicitly disable the timer.
T0	[8]	Timer 0 Present 0x0: No timer 0 0x1: Timer 0 present
T1	[9]	Timer 1 Present 0x0: No timer 1 0x1: Timer 1 present
RTC	[10]	64-bit RTC Configuration 0x0: 64-bit RTC is disabled. 0x1: 64-bit RTC is enabled.
RESERVED	[15:11]	Reserved. Read-as-zero; ignored on write.
P0	[19:16]	Indicates the interrupt priority level of Timer 0 Note: If Timer 0 is not included, this field is always set to 0.
P1	[23:20]	Indicates the interrupt priority level of Timer 1 Note: If Timer 1 is not included, this field is always set to 0.

Table 29-6 **TIMER_BUILD Field Descriptions (Continued)**

Field	Bit	Description
RESERVED	[31:24]	Reserved. Read-as-zero; ignored on write.

30

Multi-core Cluster

30.1 Multi-core Cluster Introduction

The ARC HS processor is a cluster based multi-core processor that can include ARC HS cores of similar configuration or different configuration. You can support up to eight cores in one cluster.

In a multi-core cluster ARC HS processor, the following components are available at the cluster level and shared by all the cores: Cluster Shared Memory (CSM), Stream Transfer Unit (STU), L2 cache, Shared Coherency Unit (SCU), and Bus Interface Unit (BIU).

The cluster configuration is provided in the [CLUSTER Build Configuration Register, CLUSTER_BUILD](#).

[Table 22-1](#) lists the auxiliary registers that are at a cluster level in a synchronous multi-core cluster configuration.

Table 30-1 Auxiliary Registers at the Cluster Level

Auxiliary Registers	Auxiliary Registers Addresses	Description
AUX_VOLATILE	AUX_VOLATILE (0x5E)	In a single-core configuration, this register is present in a core's auxiliary space. In a multi-core configuration, this register is physically present in the cluster auxiliary space, and is shared by all the cores.
DMP_PER_AUX	DMP_PER_AUX (0x20A)	In a single-core configuration, this register is present in a core's auxiliary space. In a multi-core configuration, this register is physically present in the cluster auxiliary space, and is shared by all the cores.
DATA_MEM_ATTR	0x20B	This register is shared by all the cores.
CSM auxiliary registers	All CSM registers	All CSM registers are shared by all the cores.
BIU Auxiliary registers	0x500 ~ 0x53F; all BIU registers	The BIU auxiliary registers are physically located in the BIU, and these registers are shared by all the cores.

Table 30-1 Auxiliary Registers at the Cluster Level

Auxiliary Registers	Auxiliary Registers Addresses	Description
PDM auxiliary registers	■ PDM_CC_PMODE (0x620) ■ PDM_CC_PSTAT(0x621) ■ PDM_CSM_PMODE(0x622) ■ PDM_CSM_PSTAT(0x623) ■ DVFS_CLUSTER_PL(0x626)	The cluster-level PDM auxiliary registers are physically located in the cluster top always on power domain and shared by all the cores. The core-level PDM registers are private to each core: PDM_PSTAT (0x610), RTT_PDM_PSTAT(0x611), PDM_PMODE(0x613), PDM_SCU_DIS_IND (0x614)
L2 Cache auxiliary registers	0x900 ~ 0x93F; all L2 cache registers	The L2 cache auxiliary registers are physically located in the L2 cache, and these registers are shared by all the cores.
Cluster DMA registers	All DMA server register (DMA_S_*)	All DMA server auxiliary registers are shared amongst the cores
Multi-core NMI registers	All multi-core NMI registers (CC_NMI_*)	These registers are shared by all the cores.
STU auxiliary registers	■ STU_COH_ENABLE0(0x98C) ■ STU_COH_AP0_BASE_L(0x98D) ■ STU_COH_AP0_SIZE(0x98F) ■ STU_COH_ENABLE1(0x990) ■ STU_COH_AP1_BASE_L ■ STU_COH_AP1_SIZE (0x993) ■ STU_DISABLE_RECORD (0x995) ■ STU_BUILD (0xE4)	These STU registers are shared by all the cores.
	0x980 to 0x98B, 0x994, 0x996, and the STU memory-mapped registers	These STU registers are private to each core.

30.2 Cluster Auxiliary and Build Configuration Registers

Table 30-2 Cluster Registers

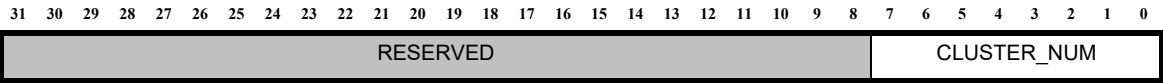
Address	Auxiliary Register Name	Description
0x298	CLUSTER_ID	Cluster ID register
0x9A9	GLOBAL_CLK_GATE_DIS	Global clock disable register
0x940	CC_NMI_STATUS	Cluster NMI status register
0x941	CC_NMI_CAUSE	Cluster NMI cause register
0xCF	CLUSTER_BUILD	Cluster build configuration register

30.2.1 CLUSTER ID Register, CLUSTER_ID

Address: 0x298

Access: r

Figure 30-1 CLUSTER_ID Register



This register contains an 8 bit cluster identification number that is assigned using the input signal: `clusternum[7:0]`.

Table 30-3 CLUSTER_ID Register

Field	Bit	Description
CLUSTER_NUM	[7:0]	Contains the cluster identification number.

30.2.2 Global Clock Disable Register, GLOBAL_CLK_GATE_DIS

Address: 0x9A9

Access: RW

Reset: 0000_0F8F

Figure 30-2 GLOBAL_CLK_GATE_DIS Register

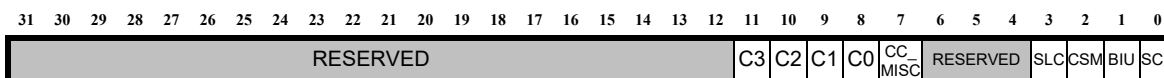
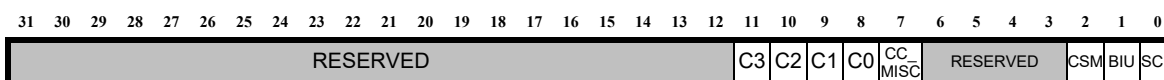


Figure 30-3 GLOBAL_CLK_GATE_DIS Register



This register is included in the processor only when the processor includes the architectural clock gating component (see CLUSTER_BUILD[30]). Use this register to disable the level 1 clock gating for the cluster components as well as the individual cores. By default, the level 1 clock gating is disabled. For performance-insensitive applications, software can program the corresponding bits of this register to enable the automatic level 1 clock gating to reduce dynamic power. For performance-sensitive applications, software can disable the automatic level 1 clock gating to get the fastest response for incoming transactions. Note that this register only controls whether the hardware must use the level 1 clock gating feature. The control signals to actually enable or disable level 1 clocks are controlled by the hardware based on the idle or busy status of each component.



Note

The TD bit affects the Level 1 clock gating function for the CPU Core. The default value (0) of the TD bit defeats the Level 1 clock gating function of the core such that the Level 1 (main) clock to the core remains on during core idle periods (halt, sleep). In this case, the timer continues to count during idle periods. When the TD bit is set to 1, the Level 1 core clock can be turned off during idle periods as controlled by GLOBAL_CLK_GATE_DIS register and the idle state of the core. For more information about the Timer TD, bit, see [“Timer 0 Control Register, CONTROL0”](#).

Table 30-4 GLOBAL_CLK_GATE_DIS Register

Field	Bit	Description
SCU	[0]	Enable dynamic clock gating at level 1 for Shared Coherency Unit (SCU) 0x1: disable 0x0: enable

Table 30-4 GLOBAL_CLK_GATE_DIS Register

Field	Bit	Description
BIU	[1]	Enable dynamic clock gating at level 1 for Bus Interface Unit (BIU) 0x1: disable 0x0: enable
CSM	[2]	Enable dynamic clock gating at level 1 for Cluster Shared Memory (CSM) 0x1: disable 0x0: enable
SLC	[3]	Enable dynamic clock gating at level 1 for System Level Cache (SLC) 0x1: disable 0x0: enable
CC_MISC	[7]	Enable dynamic clock gating at level 1 for cluster PD1 wrapper components including shared memory configuration, direct IO coherency, DMA engine (Cluster DMA or Streaming Transfer Unit), and cc_aux ■ 0x0: enable ■ 0x1: disable
C0	[8]	Enable dynamic clock gating at level 1 for CPU core 0 ■ 0x0: enable ■ 0x1: disable
C1	[9]	Enable dynamic clock gating at level 1 for CPU core 1 ■ 0x0: enable ■ 0x1: disable
C2	[10]	Enable dynamic clock gating at level 1 for CPU core 2 ■ 0x0: enable ■ 0x1: disable
C3	[11]	Enable dynamic clock gating at level 1 for CPU core 3 ■ 0x0: enable ■ 0x1: disable

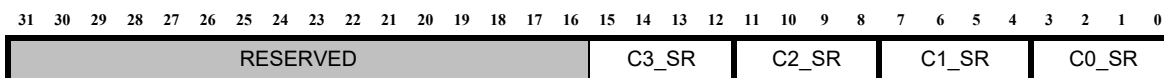
30.2.3 Cluster VCCM QoS Register, CLUSTER_VCCM_QoS

Address: 0x9AA

Access: RW

Reset: 0x000

Figure 30-4 CLUSTER_VCCM_QoS Register



This register is used to specify the service rate that each core, in the configuration, grants to the combined STU/external VCCM DMI access. This register is an optional register and exists when either STU or external VCCM DMI is configured.

Table 30-5 CLUSTER_VCCM_QoS Register

Field	Bit	Description
C0_SR	[3:0]	Specify increment step (ACCOUNT_INCR) of Core0 to serve the request from CC (combined STU/VCCM_DMI) 0x0: 50% service rate of VCCM bandwidth (account reaches zero every other cycle) (by default) 0x1: 33% service rate of VCCM bandwidth (account reaches zero every third cycle) 0x2: 25% service rate of VCCM bandwidth (account reaches zero every fourth cycle)
C1_SR	[7:4]	Specify increment step (ACCOUNT_INCR) of Core1 to serve the request from CC (combined STU/VCCM_DMI) 0x0: 50% service rate of VCCM bandwidth (account reaches zero every other cycle) (by default) 0x1: 33% service rate of VCCM bandwidth (account reaches zero every third cycle) 0x2: 25% service rate of VCCM bandwidth (account reaches zero every fourth cycle)

Table 30-5 CLUSTER_VCCM_QoS Register

Field	Bit	Description
C2_SR	[11:8]	Specify increment step (ACCOUNT_INCR) of Core2 to serve the request from CC (combined STU/VCCM_DMI) 0x0: 50% service rate of VCCM bandwidth (account reaches zero every other cycle) (by default) 0x1: 33% service rate of VCCM bandwidth (account reaches zero every third cycle) 0x2: 25% service rate of VCCM bandwidth (account reaches zero every fourth cycle)
C3_SR	[15:12]	Specify increment step (ACCOUNT_INCR) of Core3 to serve the request from CC (combined STU/VCCM_DMI) 0x0: 50% service rate of VCCM bandwidth (account reaches zero every other cycle) (by default) 0x1: 33% service rate of VCCM bandwidth (account reaches zero every third cycle) 0x2: 25% service rate of VCCM bandwidth (account reaches zero every fourth cycle)

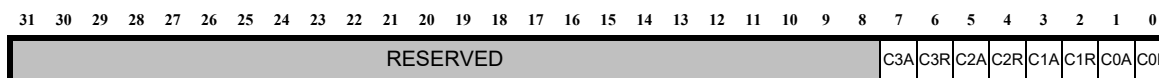
30.2.4 Cluster NMI Status Register, CC_NMI_STATUS

Address: 0x940

Access: RW

Reset: 0x0000_0000

Figure 30-5 CC_NMI_STATUS Register



You can read this register to determine which cores in the cluster have received and acknowledged the most recent request. This register exists in the processor only when `-nmi_option==true` and the processor includes a multi-core cluster. This register is shared by all the cores in a cluster. This register is updated when an NMI request is acknowledged.

The CC_NMI_STATUS register is updated to reflect the status of the most recent acknowledged NMI. If needed, NMI activity can be tracked by clearing the contents of this register to reset value and then polling this register for any change.

NMI cannot be asserted for any core that is halted or powered-down.

Table 30-6 CC_NMI_STATUS Register

Field	Bit	Description
C0R	[0]	Whether the NMI request has been asserted for Core 0. 0: asserted 1: not asserted
C0A	[1]	Whether the NMI request has been acknowledged for Core 0. 0: not acknowledged 1: acknowledged
C1R	[2]	Whether the NMI request has been asserted for Core 1. 0: asserted 1: not asserted
C1A	[3]	Whether the NMI request has been acknowledged for Core 1. 0: not acknowledged 1: acknowledged
C2R	[4]	Whether the NMI request has been asserted for Core 2. 0: asserted 1: not asserted

Table 30-6 CC_NMI_STATUS Register

Field	Bit	Description
C2A	[5]	Whether the NMI request has been acknowledged for Core 2. ■ 0: not acknowledged ■ 1: acknowledged
C3R	[6]	Whether the NMI request has been asserted for Core 3. ■ 0: asserted ■ 1: not asserted
C3A	[7]	Whether the NMI request has been acknowledged for Core 3. ■ 0: not acknowledged ■ 1: acknowledged
Reserved	[31:8]	Read as zero, ignored on writes.

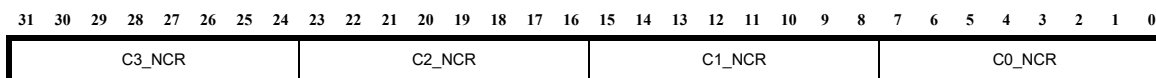
30.2.5 Cluster NMI Cause Register, CC_NMI_CAUSE

Address: 0x941

Access: R

Reset 0x0000_0000

Figure 30-6 CC_NMI_CAUSE Register



This register exists in the processor only when `-nmi_option==true`, and the processor includes a multi-core cluster.

This register provides an enumerated cause code (ENUM CODE), a snapshot of the operating mode and core state of each core for the recently acknowledged NMI request. Each core updates this register when an NMI assertion is acknowledged.

Table 30-7 CC_NMI_CAUSE Register

Field	Bit	Description
C0_NCR	[7:0]	Cause code for core 0 (see Table 30-8)
C1_NCR	[15:8]	Cause code for core 1 (see Table 30-8)
C2_NCR	[23:16]	Cause code for core 2 (see Table 30-8)
C3_NCR	[32:24]	Cause code for core 3 (see Table 30-8)

Table 30-8 NMI Cause Codes

ENUM CODE	Operating Mode	Core State	Condition	Description
0x00	Normal	Not stalled	Normal program flow	Core is running without any issue
0x01	Normal	Stalled	Page miss refill stall	Core has stalled for MMU page refill to fetch further instructions
0x02	Normal	Stalled	Cache miss refill stall	Core stalled due to instruction cache miss
0x03	Normal	Stalled	Fetch any other Stall (includes delay slot)	No instructions were fetched but due to reasons other than corresponding to ENUM codes 0x01 and 0x02 conditions
0x08	Normal	Stalled	Micro sequencer busy	Core has stall while executing a micro-sequence (for example: Enter/Leave) but not Interrupt or Exception

Table 30-8 NMI Cause Codes

ENUM CODE	Operating Mode	Core State	Condition	Description
0x09	Normal	Stalled	Auxiliary pipe stalled	An LR/SR instruction commit stalled due to a dependency
0x0a	Normal	Stalled	Vector Engine Instruction Stall	The vector DSP unit has requested the core to stall (for any reason)
0x0b	Normal	Stalled	SYNC instruction commit is stalled	A SYNC instruction commit is stalled due to a dependency that is not yet satisfied (for example: Out-of-order execution pending, Auxiliary units (FPU, DIV) pending)
0x0c	Normal	Stalled	Break hold	A Break instruction commit is stalled due to any dependency (similar to SYNC but due to a BREAK instruction)
0x0d	Normal	Stalled	Execution any other Stall	Any unidentified execution stall that dead-locks
0x10	Normal	Stalled	Operand Dependency (RAW)	An instruction has stalled when its operand is dependent on prior instruction result
0x11	Normal	Stalled	Flag Dependency	A conditional instruction commit is stalled for a prior flag instruction
0x12	Normal	Stalled	Accumulator Dependency	Core is stalled due to an accumulator dependency
0x13	Normal	Stalled	APEX Dependency	A core with an APEX extension requests core to be stalled
0x14	Normal	Stalled	Write after write (WAW)	Core has stalled when target of two instructions are same, therefore requires them to be serialized (for example: write to same peripheral)
0x20	Normal	Stalled	Out-of-order buffer full	An out-of-order execution buffer dependent instruction commit is stalled
0x21	Normal	Stalled	All Write ports busy	Instruction execution has stalled due to shortage of result forwarding ports
0x22	Normal	Stalled	Unit Busy (FPU, DIV, DSP)	An Execution unit dependency has stalled the core
0x28	Normal	Stalled	Stack Checking implicit data	Stack Checking/MPU needs is dependent of core state
0x30	Normal	Stalled	Target data in external unit (ICCM, Vector Unit, Peripheral)	Core is dead-locked core due to an interdependent memory operation

Table 30-8 NMI Cause Codes

ENUM CODE	Operating Mode	Core State	Condition	Description
0x31	Normal	Stalled	Data CCM busy	A memory operation to the DCCM is stalled as the DCCM is busy
0x32	Normal	Stalled	Data Cache miss	A memory instruction stalled due to a Data Cache Miss
0x33	Normal	Stalled	Auxiliary Data Operation busy	Core is stalled when the auxiliary data operation is busy
0x34	Normal	Stalled	External data request	A memory instruction stalled due to a Data on external bus
0x40	Exception	Stalled	Vector Fetch	An exception cannot be taken due to a blocked vector jump (This occurs due to unresponsive fetch of vector region)
0x48	Interrupt	Stalled	Vector Fetch	An interrupt cannot be taken due to a blocked vector jump (This occurs due to unresponsive fetch of vector region)
0x49	Interrupt	Stalled	Post commit data	An interrupt cannot be taken or RTIE cannot return due to any reason
0x60	Debug	Debug killed	NMI killed Debug instruction	An NMI has terminated the debugger port (might or might not be stalled) and not due to any other debug conditions
0x61	Debug	Stalled	Post commit data	Debugger cannot be started due to pending non-memory instructions
0x62	Debug	Stalled	Instruction Fetch	Debugger cannot be started as the core is dead-locked while fetching non-debug instructions
0x63	Debug	Stalled	Any Memory	Debugger cannot be started due to pending memory instructions
0x64	Debug	Stalled	External Bus	Debugger cannot be started due to a dead-locked external bus that needs to be serviced
0x65	Debug	Stalled	Vector Unit	Debugger is waiting on the core to complete all pending out-of-order transactions that are not memory related
0x70	Sleep	Wakeup	Core is in sleep mode	NMI wakes up a core
0x80	Halt	NMI ignored	Halted or Powered down	A core is halted or powered down and does not respond to an NMI
0x81	Halt	Stalled	Exception	A halt cannot be completed due to a pending exception that must be serviced

Table 30-8 NMI Cause Codes

ENUM CODE	Operating Mode	Core State	Condition	Description
0x82	Halt	Stalled	Debug	Halt and debugger cause mutual dead-lock
0x83	Halt	Stalled	Post commit data	A dead-locked read or write memory
0x84	Halt	Stalled	Instruction fetch	A halt cannot be started as the core is dead-locked while fetching non-debug instructions
0x85	Halt	Stalled	Any Memory	A halt cannot be started due to pending memory instructions
0x86	Halt	Stalled	External Bus	A halt cannot be started due to a dead-locked external bus
0x87	Halt	Stalled	Vector Unit	A halt is waiting for the core to complete all pending out-of-order transactions that are not memory related

30.2.6 CLUSTER Build Configuration Register, CLUSTER_BUILD

Address: 0xCF

Access: R

Figure 30-7 CLUSTER_BUILD Register

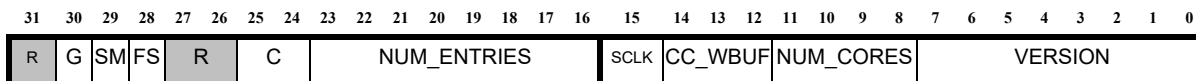
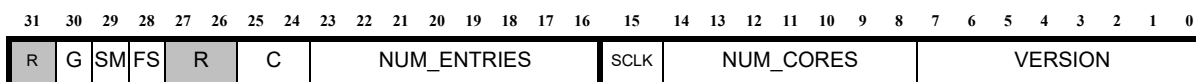


Figure 30-8 CLUSTER_BUILD Register



The overall configuration of a coherent cluster of cores is available to application through this read-only build configuration register (BCR).

This register is always present in a processor.

Table 30-9 CLUSTER_BUILD Register

Field	Bit	Description
Version	[7:0]	Cluster version. All non-enumerated values are reserved. 0x1: Initial version; Direct I/O coherency is not supported. 0x2: Second version; Direct I/O coherency is supported. 0x3: Includes the DMP peripheral configuration. 0x4: Includes support for two peripheral regions. 0x5: Indicates fine-grained clock gating control for cluster components 0x7: Indicates the presence of the BIU_BUILD register. Bits [27:26] are now reserved. The configuration of the peripheral interface is now recorded in the BIU_BUILD register. 0x8: Indicates QoS support for STU/VCCM DMI accesses to VCCM
NUM_CORES	[11:8]	Indicates the number of cores in the coherent cluster.

Table 30-9 CLUSTER_BUILD Register

Field	Bit	Description
CC_WBUF	[14:12]	Number of cluster-wide buffer entries (specified by the <code>cc_vccm_wbuf_depth</code> option) to handle STU/VCCM_DMI transactions to VCCM 0x0: No cluster-wide buffer configured (compatible with previous version) 0x1: 1 entry 0x2: 2 entries 0x3: 3 entries 0x4: 4 entries All other values are reserved
SCLK	[15]	Indicates the relationship of system clock and ARC clock (clock ratio or separate clock pins). 0x0: Support clock ratio for system clock 0x1: Support separate clock for system clock Note: This field is exists when cluster version is greater than 0x5.
NUM_ENTRIES	[23:16]	Indicates the maximum number of concurrent transactions supported by the snoop control unit in the cluster.
C	[25:24]	I/O coherency. 0x0: I/O coherency is not supported. 0x1: I/O coherency is supported through the system bus (AXI, AHB-Lite). 0x2: I/O coherency is supported through the direct I/O coherency interface.
FS	[28]	Indicates whether the cluster is built in a FS (functional safety) system or non-FS system.
SM	[29]	Specified whether the share memory configuration is enabled. 0x0: shared memory configuration between the CSM and the L2 cache is disabled. 0x1: shared memory configuration between the CSM and the L2 cache is enabled.
G	[30]	Indicates the presence of the architectural gate clocking configuration in the processor. 0x0: architectural clock gating is not configured. 0x1: architectural clock gating is configured. This bit reflects the value of the <code>-clock_gating</code> configuration option and is available only when <code>VERSION</code> of this register is greater than 0x4.

31

Shared Coherency Unit

The Shared Coherency Unit (SCU) implements cache coherency within a cluster of one to four cores. The cluster effectively defines a coherency domain, within which a consistent view of a shared physical memory space is maintained by a hardware coherency protocol. There are a number of optional components within the coherency domain, including a coherent memory target port and a unified (level-2) system-level cache. The coherent memory target port allows externally derived DMA transactions to participate in coherency protocols. The SCU provides direct support for external devices such as DMA devices to access the memory via the cache hierarchy, thereby providing coherency between I/O transactions and the complete memory hierarchy. Majority of the SCU behavior is fixed at configuration time. However, some aspects of the coherent I/O behavior can be defined by software.

[Table 31-1](#) provides an overview of the auxiliary registers that you can use to program the coherent I/O region behavior. The SCU registers are physically present in the Bus Interface Unit (BIU), and are shared by all the cores.

31.1 Auxiliary Registers

Table 31-1 I/O Coherency Programming Registers

Address	Auxiliary Register Name	Description
0x500	BIU_IO_0_COH_ENABLE	Enable coherency aperture for I/O
0x501	BIU_IO_0_COH_PARTIAL	Enable partial coherency for I/O
0x508	BIU_IO_0_COH_AP0_BASE	Base address for coherency aperture
0x509	BIU_IO_0_COH_AP0_SIZE	Size of coherency aperture
0xE7	BIU_BUILD	BIU configuration register

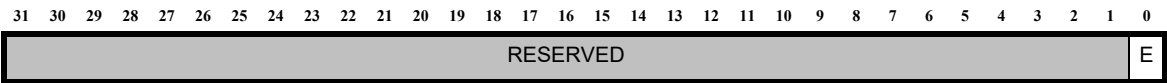
31.1.1 I/O Coherency Enable Register, BIU_IO_0_COH_ENABLE

Address: 0x500

Access: RW

Reset 0x0000 0000

Figure 31-1 BIU_IO_0_COH_ENABLE Register



This register is included in the processor build only if `-biu_ioc_bus_num==1`. This register was named `IO_0_COH_ENABLE` in the earlier releases.

The `BIU_IO_0_COH_ENABLE` register is used to enable or disable input or output (I/O) coherency snooping within the shared coherency unit (SCU). There is a single target port, which may be interfaced to any of the bus types supported by the cluster’s bus interface unit (BIU).

To enable I/O coherency snooping, set the enable bit `E` to 1. When snooping is disabled, all I/O transactions going through the coherent I/O port do not have any effect on cached copies of locations accessed by the I/O transactions within the cluster.

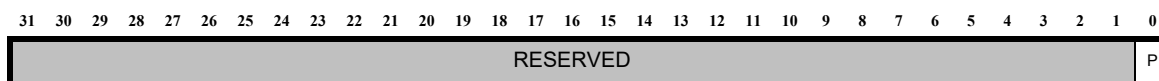
Table 31-2 BIU_IO_0_COH_ENABLE Bit Field Description

Field	Bit	Description
E	[0]	Enable or disable I/O coherency snooping within the shared coherency unit (SCU). 0: Disable snooping 1: Enable snooping

31.1.2 I/O Coherency Partial Write Register, BIU_IO_0_COH_PARTIAL

Address: 0x501
 Access: RW
 Reset 0x0000 0001

Figure 31-2 BIU_IO_0_COH_PARTIAL Register



This register is included in the processor build only if `-biu_ioc_bus_num==1`. This register was named `IO_0_COH_PARTIAL` in the earlier releases.

Writing to the `BIU_IO_0_COH_PARTIAL` register allows the programmer to define whether the corresponding I/O transaction that triggers the write snoop requests always snoops full cache lines or not. If the `P` bit is not set, all write transactions redefine a full cache line; any processor containing that cache line need not writeback a dirty cached copy prior to the I/O write taking place. If the `P` bit is set, I/O writes may partially overwrite cache lines. Therefore, any processor holding a dirty cached copy must writeback such cache lines before the I/O write is allowed to proceed. The `P` bit is set to 1 on reset.

Table 31-3 BIU_IO_0_COH_PARTIAL Bit Field Description

Field	Bit	Description
P	[0]	Coherent I/O writes may write partial lines. ■ 0: Writes always write a complete cache line ■ 1: Writes may write a partial cache line

31.1.3 I/O Coherency Address Aperture Base Address, BIU_IO_0_COH_APO_BASE

Address: 0x508

Access: RW

Reset 0x0000 0000

Figure 31-3 BIU_IO_0_COH_APO_BASE Register when Physical Address Extension is Disabled

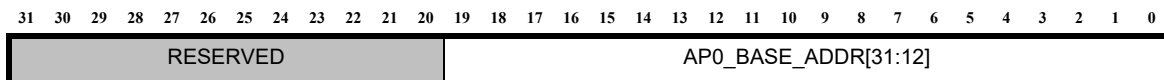
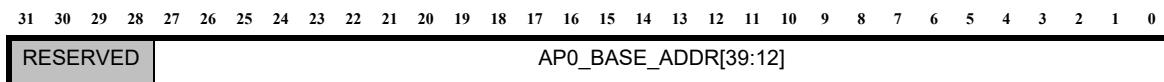


Figure 31-4 BIU_IO_0_COH_APO_BASE Register when Physical Address Extension is Enabled



This register is included in the processor build only if `-biu_ioc_bus_num==1`. This register was named `IO_0_COH_APO_BASE` in the earlier releases.

The `BIU_IO_0_COH_APO_BASE` register specifies the base physical address of the address aperture to be snooped. All snoop requests outside the aperture defined by the `BIU_IO_0_COH_APO_BASE` and `BIU_IO_0_COH_APO_SIZE` registers are not checked for I/O coherency. This base address is aligned to a 4 KB physical address boundary. Therefore, it does not include the lower 12 bits of the physical address.

When the physical address extension option is disabled, bits [19:0] of the base address register correspond to bits [31:12] of the start of the coherent region, and the upper 12 bits are ignored on writes and read as zero. When the physical address extension option is enabled, bits [27:0] of the base address register correspond to bits [39:12] of the start of the region, and the upper 4 bits are ignored on writes and read as zero.

The base address of the I/O coherency region is therefore always aligned to a 4 KB boundary, and in addition must also be aligned to the size of the aperture as specified in the `BIU_IO_0_COH_APO_SIZE` register. The coherency hardware forcibly aligns any base address to the size of the coherency aperture by ignoring the irrelevant low-order bits in the `APO_BASE_ADDR` field.

Table 31-4 BIU_IO_0_COH_APO_BASE Bit Field Description

Field	Bit	Description
APO_BASE_ADDR	[19:0] if physical address extension is disabled. [27:0] if physical address extension is enabled.	Specifies the physical base address of the address aperture to be snooped. This register holds bits [31:12] of the physical address of the start of the I/O coherency region (or bits 39:12 when 40-bit physical address extension are enabled in the configuration).

31.1.4 I/O Coherency Aperture Size, BIU_IO_0_COH_AP0_SIZE

Address: 0x509

Access: RW

Reset 0x0000 0000

Figure 31-5 BIU_IO_0_COH_AP0_SIZE Register when Physical Address Extension is Disabled

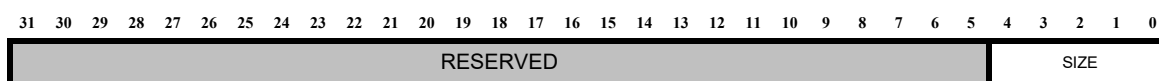
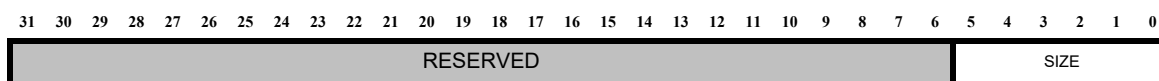


Figure 31-6 BIU_IO_0_COH_AP0_SIZE Register when Physical Address Extension is Enabled



This register is included in the processor build only if `-biu_ioc_bus_num==1`. This register was named `IO_0_COH_AP0_SIZE` in the earlier releases.

Writing to the `BIU_IO_0_COH_AP0_SIZE` register allows the programmer to set the size of the address aperture within which transactions on the I/O coherency bus are kept coherent with cached data in the coherency domain of the multi-core cluster. The effective aperture size is $2^{(SIZE+2)}$ KB. A `SIZE` of 0 therefore specifies an aperture of 4 KB, and a `SIZE` of 1 specifies the aperture size as 8 KB.

Due to the offset of 2, it is possible to specify aperture sizes larger than the normal physical address space (4 GB). Any value assigned to the `SIZE` field that would place the upper bound of the aperture beyond the highest physical address sets the highest physical address as the upper bound of the aperture. This occurs for `SIZE > 20` when the physical address extension option is disabled, or `SIZE > 28` when the physical address extension option is enabled.

Table 31-5 BIU_IO_0_COH_AP0_SIZE Bit Field Description

Field	Bit	Description
SIZE	[4:0] if the physical address extension is disabled. [5:0] if the physical address extension is enabled.	When the physical address extension is disabled: - SIZE values in the range 0 to 20 represent coherency aperture sizes of $2^{(SIZE+2)}$. - SIZE values in the range 21 to 31 represent coherency apertures of 4GB (the full physical address space). When the physical address extension is enabled: - SIZE values in the range 0 to 28 represent coherency aperture sizes of $2^{(SIZE+2)}$. - SIZE values in the range 29 to 63 represent coherency apertures of 1TB (the full physical address space).

31.1.5 Bus Interface Unit Build Configuration Register, BIU_BUILD

Address: 0xE7

Access: R

Reset Build-time configurable

Figure 31-7 BIU_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
R	SCL_K	P	ECC_D	ECC	R	UP	UC	UI	UB	DMI_BUS	IOC_BUS	PER_BU S	MEM_BU S	Version																	

Figure 31-8 BIU_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
R	SCL	P	ECC_D	ECC	R	UP	UC	UI	UB	VD	DMI_BUS	IOC_BUS	PER_BU	MEM_BU	Version																

This register provides the configuration of the BIU. If BIU is not configured in your processor, this register does not exist and returns 0 on reads. This register is present only for configurations with CLUSTER_BUILD.VERSION==0x7. This register indicates the presence of some external interface attributes that might impact software programming: presence of specific bus interface (memory, peripheral, slave, and so on), the number of specific bus interface (memory, peripheral, slave, and so on) and the bus-ecc protection support.

Table 31-6 BIU_BUILD Register

Field	Bit	Description
Version	[7:0]	0x0: Initial version 0x1: Indicates BIU_BUILD version 1 0x2: Indicates separate VCCM DMI configuration All other values are reserved.
MEM_BUS	[9:8]	Indicates the supported BIU memory bus initiators as specified by the option <code>biu_mem_bus_num</code> . 0x0: No memory initiator bus 0x1: 1 combined memory initiator bus (CBU) 0x2: 2 separate memory initiator buses (DBU,DBU) 0x3: Dedicated memory buses per core

Table 31-6 BIU_BUILD Register

Field	Bit	Description
PER_BUS	[11:10]	Indicates the number of supported peripheral buses. 0x0: no DMP peripheral interface in the cluster. 0x1: Single peripheral interface (dmp_peripheral0) and dedicated peripheral bus is disabled in multi-core (biu_per0_dedicated_bus=0). 0x2: Two peripheral interfaces (dmp_peripheral0/dmp_peripheral1) and dedicated peripheral bus is disabled in multi-core (biu_per0_dedicated_bus=0 and biu_per1_dedicated_bus=0). 0x3: Dedicated peripheral bus is enabled in multi-core cluster (biu_per0_dedicated_bus=1 or biu_per1_dedicated_bus=1).
IOC_BUS	[14:12]	Indicates the number of external IOC bus numbers. 0x0: No IOC bus (biu_udmi_bus_num=1 biu_ioc_bus_num = 0) 0x1: 1 IOC bus (biu_udmi_bus_num=0 & biu_ioc_bus_num=1) All other values are reserved.
DMI_BUS	[17:15] [16:15]	Indicates the number of BIU DMI buses. 0x0: No DMI bus (-biu_dmi_bus_num=0 or no ICCM&DCCM) 0x1: 1 combined DMI bus SCCM (biu_dmi_bus_num=1 or biu_udmi_bus_num 1) 0x2: 2 separate DMI buses (each for ICCM and DCCM) (biu_dmi_bus_num=2) 0x3: 1 combined DMI bus and 1 DMI bus for VCCM 0x4: 2 separate DMI buses for ICCM and DCCM, and 1 DMI bus for VCCM All other values are reserved.
VCCM_DMI_BUS_NUM (VD)	[17]	- BIU DMI bus number for accessing VCCM 0x0: No VCCM DMI interface 0x1: Has VCCM DMI bus interface
USLV_BUS (UB)	[18]	Indicates if the unified slave port is enabled or disabled. Reflects the value of the biu_udmi_bus_num option . 0x0: Unified slave port is disabled 0x1: Unified slave port is enabled
USLV_SUPPORT_IOC (UI)	[19]	Indicates the BIU unified slave port access to the coherent space. Reflects the value of the biu_udmi_support_ioc option . 0x0: Unified slave port can not access coherent space 0x1: Unified slave port can access coherent space

Table 31-6 BIU_BUILD Register

Field	Bit	Description
USLV_IOC_FULL_COH (UC)	[20]	Indicates the BIU unified slave port IOC region transactions. Reflects the value of the <code>biu_udmi_ioc_full_coherent</code> option. 0x0: Accesses through the unified slave port IOC region can be coherent or non-coherent transaction 0x1: Accesses through the unified slave port IOC region are all coherent transactions
USLV_ACCESS_PER1 (UP)	[21]	Indicates whether the unified slave port can access the peripheral interface 1. 0x0: Unified slave port cannot access peripheral interface 1 0x1: Unified slave port can access peripheral interface 1
BUS_ECC_EN (ECC)	[25:24]	The BIU bus ecc protection enable bits. 0x0: ECC on bus interface is disabled (<code>cc_support_bus_ecc=0</code>) 0x1: ECC on bus interface is enabled (<code>cc_support_bus_ecc=1</code>) without all 0/1 protection (<code>cc_bus_ecc_all_01=0</code>) 0x2: All zeros and ones detection is enabled (<code>cc_support_bus_ecc=1</code>) with all 0/1 protection (<code>cc_bus_ecc_all_01=1</code>) 0x3: Reserved
BUS_ECC_DATA_OPTION (ECC_D)	[27:26]	The BIU bus ecc data protection type. Reflects the value of the <code>cc_bus_ecc_data_option</code> option. 0x0: SECDDED on 32-bit data 0x1: SECDDED on 64-bit data 0x2 to 0x3: Reserved
BUS_ECC_PARITY_OPTION (P)	[28]	The BIU bus ecc parity protection type. Reflects the value of the <code>cc_bus_ecc_parity_option</code> option. 0x0: Even parity on control signals for AXI buses 0x1: Even parity on control signals for AXI buses
SCLK	[29]	Indicates the relationship of BIU clock and ARC clock 0x0: Supports only clock ratio 0x1: Support for separate BIU clock

32

Power Management Features

32.1 Introduction to Power Management Features

The ARCV2 ISA provides a number of features to control power consumption, depending on the configuration of the core. These features may include power domain management (PDM), to reduce static (leakage) power through the power-down of unused components, features to enable an external controller to dynamically adjust the supply voltage and operating frequency of the core (DVFS), and features to limit the maximum instruction execution rate of the core to reduce dynamic power (execution control).

- [PDM and DVFS Register Interface](#)
- [Execution Rate Interface](#)

32.2 PDM and DVFS Register Interface

Table 32-1 Power Domain Management Registers

Address	Name	Description
0x610	Core Power Status Register, PDM_PSTAT	Core power status; this register is present only when PDM is configured.
0x611	ARC Trace Power Status Register, RTT_PDM_PSTAT	ARC RTT power status; this register is present only when ARC RTT and PDM are configured.
0x613	Power Down Register, PDM_PMODE	Core power down programming register; this register is present only when PDM is configured.
0x614	Core Snoop Indicator Control Register, PDM_SCU_DIS_IND	Core Snoop Indicator Control Register
0x620	Power Down Cluster Register, PDM_CC_PMODE	Cluster power-down programming register
0x621	Cluster Power Status Register, PDM_CC_PSTAT	Cluster power status register

Table 32-1 Power Domain Management Registers

Address	Name	Description
0x622	Power Down Cluster Shared Memory Register, PDM_CSM_PMODE	CSM power-down programming register
0x623	Cluster CSM Power Status Register, PDM_CSM_PSTAT	CSM power status register
0x624	Power Down L2 Cache Register, PDM_SLC_PMODE	Power Down L2 Cache Register,
0x625	Cluster L2 Cache Power Status Register, PDM_SLC_PSTAT	L2 cache power status register
0x626	Cluster Performance Register, DVFS_CLUSTER_PL	Current performance level of the cluster
0x627	Partial Power-Down Cluster Shared Memory Register, PDM_CSM_P_PMODE	Partial Power-Down Cluster Shared Memory Register
0x628	Partial Power-Down Cluster Shared Memory Register, PDM_CSM_P_PSTAT	Partial Power-Down Cluster Shared Memory Status Register
0x629	Partial Power-Down L2 Cache Register, PDM_SLC_P_PMODE	Partial Power-Down L2 Cache Register,
0x62A	Partial Power-Down L2 Cache Register, PDM_SLC_P_PSTAT	Partial Power-Down L2 Cache Status Register
0x62B	Shared Memory Power Down Register, PDM_CSM_SM_P_CFG	CSM shared memory partial power down granularity configure register
0xF7	Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD	Build configuration register

32.2.1 Core Power Status Register, PDM_PSTAT

Address: 0x610

Access: RW

Reset: 0x0000_0000

Figure 32-1 PDM_PSTAT Register

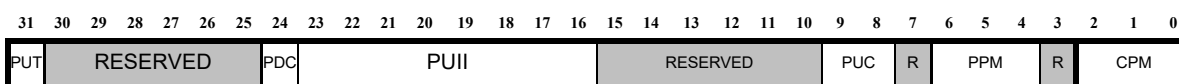
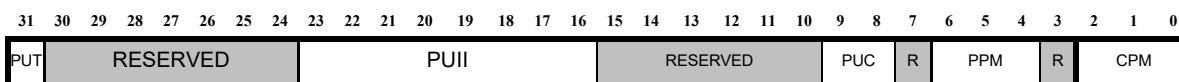


Figure 32-2 PDM_PSTAT Register



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`.

In a single-core processor configuration, this register always exists in the core auxiliary space. In a multi-core processor implementation, this register exists in the cluster auxiliary space.

Table 32-2 PDM_PSTAT Field Description

Field	Bit	Reset	Description
CPM	[2:0]	0	Current power mode of the core. Read-only. This field can only be controller by the hardware. Write to this field are ignored. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode For more information about the power-down modes, see Table 32-4 on page 1246.
PPM	[6:4]	0	Previous power mode of the core. Read-only. This field can only be controller by the hardware. Write to this field are ignored. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode For more information about the power-down modes, see Table 32-4 on page 1246.

Table 32-2 PDM_PSTAT Field Description

Field	Bit	Reset	Description
PUC	[9:8]	0	Power up cause. Read-only. This field is relevant only if the core was previously in a power-down mode. 0x0: Power up by interrupt; for more information about the interrupts that can power up, see the <i>DesignWare ARCV2-based processor Databook</i>. 0x1: Power up by debugger 0x2: Power up by an external event This field can only be controlled by hardware. Writes to this field are ignored.
PUII	[23:16]	0	This field is relevant only if the core was previously in a power-down mode and was powered up by an interrupt. The index of the power up interrupt. Value range is from 16 to 255. Read-only.
PDC	[24]	0	Power down cause. 0x0: power down by PMODE register. 0x1: power down by external event
PUT	[31]	0	Power up trigger bit. This bit is write-only and is always read as zero. - Write 1: - When core is in power-down state: Send power up request to the PMU and halt the core after PMU has powered up the core. - When core is in power on state: halt the core. - Write 0: No effect; ignored.

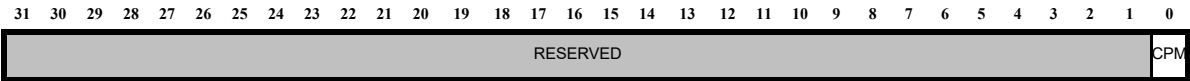
32.2.2 ARC Trace Power Status Register, RTT_PDM_PSTAT

Address: 0x611

Access: RW

Reset: 0x0000_0000

Figure 32-3 RTT_PDM_PSTAT Register



Use this register to power down or power up ARC Trace. Reading this register returns the current ARC RTT power status.

In a single-core processor configuration, this register always exists in the core auxiliary space. In a multi-core processor implementation, this register exists in the cluster auxiliary space.

Table 32-3 RTT_PDM_PSTAT Field Description

Field	Bit	Reset	Description
CPM	[0]	0	Write to this bit to power down or power up ARC Trace ■ 0x0: Send ARC Trace power up request to the external PMU. ■ 0x1: Send ARC Trace power down ARC Trace to the external PMU in PM1 mode. Read this bit to determine the power status of ARC Trace: ■ 0x0: The external PMU has powered up ARC Trace. ■ 0x1: The external PMU has powered down ARC Trace.

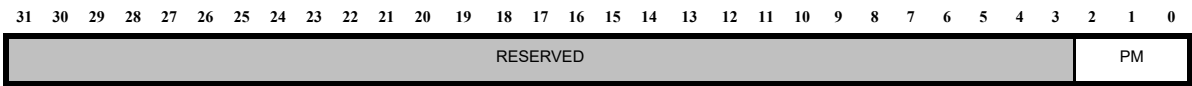
32.2.3 Power Down Register, PDM_PMODE

Address: 0x613

Access: W

Default: 0x0000_0000

Figure 32-4 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.

In a single-core processor configuration, this register always exists in the core auxiliary space. In a multi-core processor implementation, this register exists in the cluster auxiliary space.

For information about the power domains, see the *DesignWare ARCV2-based Processor Databook*.

Table 32-4 PDM_PMODE Field Description

Field	Bit	Reset	Description
PM	[2:0]	0	Power-down mode 0x0: reserved 0x1: power-down mode 1 (PM1) 0x2: power-down mode 2 (PM2)

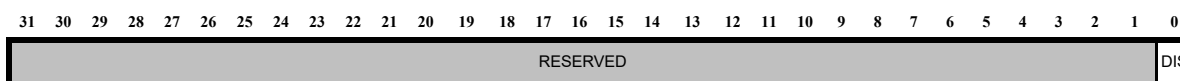
32.2.4 Core Snoop Indicator Control Register, PDM_SCU_DIS_IND

Address: 0x614

Access: RW

Default: 0x0000_0000

Figure 32-5 PDM_SCU_DIS_IND Register



Use this register to inform the shared coherency unit (SCU) that the current core cannot be snooped.

This register is present only if the shared coherency unit is configured in the core cluster. Also, this register is private to each core.

After software programs this register, the software must poll the register value. The programming takes effect only when the register is updated.

Table 32-5 PDM_SCU_DIS_IND Field Description

Field	Bit	Reset	Description
DSI	[0]	0	Core enable/disable snoop indicator 0x0: SCU can snoop the core. 0x1: SCU cannot snoop the core.

Note: This register is only used when an external PMU directly controls the core power down and core power up (that is, you do not use the PMODE register to power down or power up a core). Use the following sequence to program this register:

32.2.5 Disable SCU Transaction

1. Flush the data cache if dirty bits exist in the data cache.
2. Invalidate the data cache.
3. Program the PDM_SCU_DIS_IND register to disable the SCU snoop transactions.
4. Start the power-down sequence.

32.2.6 Enable SCU Transaction

1. Start the power-up sequence.
2. After the core is powered up, enable the data cache if necessary.
3. Program the PDM_SCU_DIS_IND register to enable SCU snoop transaction.

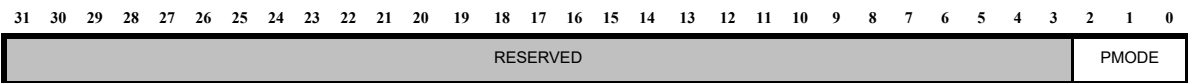
32.2.7 Power Down Cluster Register, PDM_CC_PMODE

Address: 0x620

Access: RW

Default: 0x0000_0000

Figure 32-6 PDM_CC_PMODE Register



Use this register to program the power-down mode of a multi-core cluster. This register is not cleared to 0 after the cluster is powered down; it retains the value written by the software.

This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

For information about the power domains, see the *DesignWare ARCV2-based Processor Databook*.

Table 32-6 PDM_CC_PMODE Field Description

Field	Bit	Reset	Description
PMode	[2:0]	0	Power-down mode 0x0: the cluster is in power-on mode when all the cores are powered-down. 0x1: the cluster enters power-down mode 1 (PM1) when all the cores are powered-down. 0x2: the cluster enters power-down mode 2(PM2) when all the cores are powered-down. Other values are reserved

32.2.8 Cluster Power Status Register, PDM_CC_PSTAT

Address: 0x621

Access: R

Reset: 0x0000_0000

Figure 32-7 PDM_CC_PSTAT Register for a Four-Core Cluster

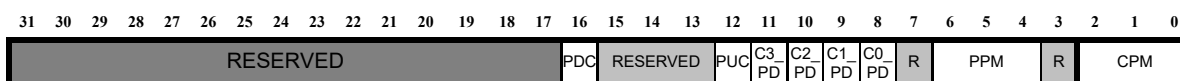
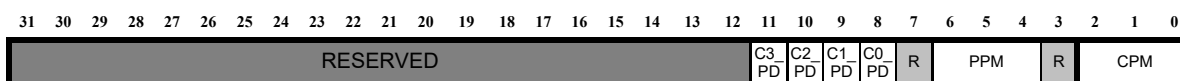


Figure 32-8 PDM_CC_PSTAT Register for a Four-Core Cluster



This register indicates the current power mode of a multi-core cluster, the previous power mode of the cluster, and the current power mode of each of the cores in the cluster.

This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

Table 32-7 PDM_CC_PSTAT Field Description

Field	Bit	Description
CPM	[2:0]	Current power mode of the multi-core cluster. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode For more information about the power-down modes, see Table 32-6 on page 1248.
PPM	[6:4]	Previous power mode of the multi-core cluster. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode For more information about the power-down modes, see Table 32-6 on page 1248.
C0_PD	[8]	Power status of core 0. Read-only. 0x0: Core 0 is currently powered on. 0x1: Core 0 is currently powered down.

Table 32-7 PDM_CC_PSTAT Field Description

Field	Bit	Description
C1_PD	[9]	Power status of core 1. Read-only. 0x0: Core 1 is currently powered on. 0x1: Core 1 is currently powered down. If the core 1 is not present in the processor. Reads to this bit return 0.
C2_PD	[10]	Power status of core 2. Read-only. 0x0: Core 2 is currently powered on. 0x1: Core 2 is currently powered down. If the core 2 is not present in the processor. Reads to this bit return 0.
C3_PD	[11]	Power status of core 3. Read-only. 0x0: Core 3 is currently powered on. 0x1: Core 3 is currently powered down. If the core 3 is not present in the processor. Reads to this bit return 0.
PUC	[12]	Power up cause. 0x0: Power up by cluster core power up 0x1: Power up by an external event
PDC	[16]	Power down cause. 0x0: Power down by PMODE register. 0x1: Power down by an external event

Table 32-8 PDM_CC_PSTAT Field Description

Field	Bit	Description
CPM	[2:0]	Current power mode of the multi-core cluster. Read-only. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode For more information about the power-down modes, see Table 32-6 on page 1248.
PPM	[6:4]	Previous power mode of the multi-core cluster. Read-only. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode For more information about the power-down modes, see Table 32-6 on page 1248.

Table 32-8 PDM_CC_PSTAT Field Description

Field	Bit	Description
C0_PD	[8]	Power status of core 0. Read-only. 0x0: core 0 is currently powered on. 0x1: core 0 is currently powered down.
C1_PD	[9]	Power status of core 1. Read-only. 0x0: core 1 is currently powered on. 0x1: core 1 is currently powered down. If the core 1 is not present in the processor. Reads to this bit return 0.
C2_PD	[10]	Power status of core 2. Read-only. 0x0: core 2 is currently powered on. 0x1: core 2 is currently powered down. If the core 2 is not present in the processor. Reads to this bit return 0.
C3_PD	[11]	Power status of core 3. Read-only. 0x0: core 3 is currently powered on. 0x1: core 3 is currently powered down. If the core 3 is not present in the processor. Reads to this bit return 0.

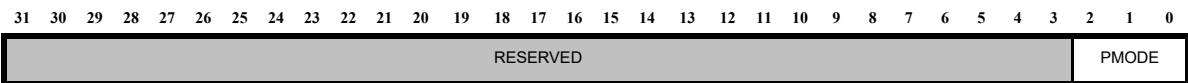
32.2.9 Power Down Cluster Shared Memory Register, PDM_CSM_PMODE

Address: 0x622

Access: RW

Default: 0x0000_0000

Figure 32-9 PDM_CSM_PMODE Register



Use this register to program the power-down mode of a cluster shared memory (CSM). This register is not cleared to 0 after CSM is powered down; it retains the value written by the software.

This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

For information about the power domains, see the *DesignWare ARCV2-based Processor Databook*.

Table 32-9 PDM_CSM_PMODE Field Description

Field	Bit	Reset	Description
PMode	[2:0]	0	Power-down mode of CSM 0x0: Set CSM power-up mode 0x1: Set CSM power-down mode 1 (PM1) 0x2: Set CSM power-down mode 2 (PM2) Other values are reserved.

32.2.10 Cluster CSM Power Status Register, PDM_CSM_PSTAT

Address: 0x623

Access: R

Reset: 0x0000_0000

Figure 32-10 PDM_CSM_PSTAT Register

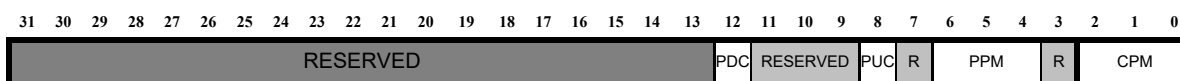
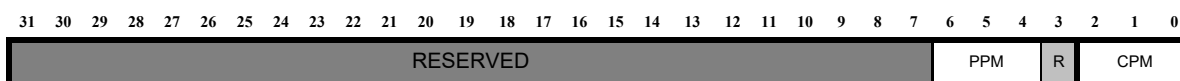


Figure 32-11 PDM_CSM_PSTAT Register



This register indicates the current power mode of CSM, the previous power mode of CSM, the power up cause, and the interrupt index that has caused the power up of the CSM.

This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

Table 32-10 PDM_CSM_PSTAT Field Description

Field	Bit	Reset	Description
CPM	[2:0]	0	Current power mode of CSM. Read-only. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode Other values are reserved.
PPM	[6:4]	0	Previous power mode of CSM. Read-only. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode Other values are reserved.
PUC	[8]	0	Power up cause. 0x0: Power up by Programming PDM_CSM_PMODE register 0x1: Power up by an external event

Table 32-10 PDM_CSM_PSTAT Field Description

Field	Bit	Reset	Description
PDC	[12]	0	Power down cause. 0x0: Power down by PMODE register. 0x1: Power down by an external event

Table 32-11 PDM_CSM_PSTAT Field Description

Field	Bit	Reset	Description
CPM	[2:0]	0	Current power mode of CSM. Read-only. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode Other values are reserved.
PPM	[6:4]	0	Previous power mode of CSM. Read-only. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode Other values are reserved.

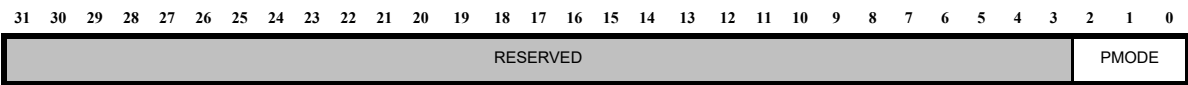
32.2.11 Power Down L2 Cache Register, PDM_SLC_PMODE

Address: 0x624

Access: RW

Default: 0x0000_0000

Figure 32-12 PDM_SLC_PMODE Register



Use this register to program the power-down mode of a the L2 cache. This register is not cleared to 0 after SLC is powered down; it retains the value written by the software.

This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

For information about the power domains, see the *DesignWare ARCv2-based Processor Databook*.

Table 32-12 PDM_SLC_PMODE Field Description

Field	Bit	Reset	Description
PMode	[2:0]	0	Power-down mode of SLC 0x0: Set SLC power-up mode 0x1: Set SLC power-down mode 1 (PM1) 0x2: Set SLC power-down mode 2 (PM2) Other values are reserved.

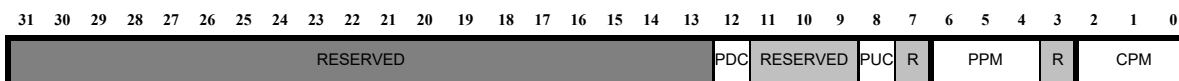
32.2.12 Cluster L2 Cache Power Status Register, PDM_SLC_PSTAT

Address: 0x625

Access: R

Reset: 0x0000_0000

Figure 32-13 PDM_SLC_PSTAT Register



This register indicates the current power mode of L2 cache, the previous power mode of L2 cache, the power up cause, and the interrupt index that has caused the power up of the L2 cache.

This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

Table 32-13 PDM_SLC_PSTAT Field Description

Field	Bit	Reset	Description
CPM	[2:0]	0	Current power mode of L2 cache. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode Other values are reserved.
PPM	[6:4]	0	Previous power mode of L2 cache. 0x0: Power on mode 0x1: PM1 mode 0x2: PM2 mode Other values are reserved.
PUC	[8]	0	Power up cause. 0x0: Power up by Programming PDM_SLC_PMODE register 0x1: Power up by an external event
PDC	[12]	0	Power down cause. 0x0: Power down by PMODE register. 0x1: Power down by an external event

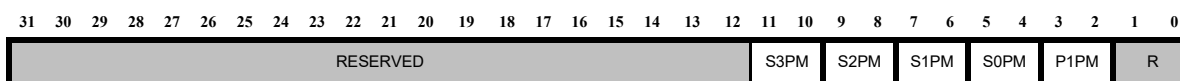
32.2.13 Partial Power-Down Cluster Shared Memory Register, PDM_CSM_P_PMODE

Address: 0x627

Access: RW

Default: 0x0000_0000

Figure 32-14 PDM_CSM_P_PMODE Register



Use this register to program the partial power-down mode of a cluster shared memory (CSM). This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

For information about the power domains, see the *DesignWare ARCV2-based Processor Databook*.

Table 32-14 PDM_CSM_P_PMODE Field Description

Field	Bit	Reset	Description
P1PM	[3:2]	0	CSM partition 1 memory power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Note: if software does not want to change its power mode, it is recommended set the value as 2'b11.
S0PM	[5:4]	0	Shared memory quadrant 0 power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if software does not want to change its power mode, it is recommended set the value as 2'b11. - if shared memory quadrant 0 does not map to CSM, set the value as 2'b11.

Table 32-14 PDM_CSM_P_PMODE Field Description

Field	Bit	Reset	Description
S1PM	[7:6]	0	Shared memory quadrant 1 power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if software does not want to change its power mode, it is recommended set the value as 2'b11. - if shared memory quadrant 1 does not map to CSM, set the value as 2'b11.
S2PM	[9:8]	0	Shared memory quadrant 2 power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if software does not want to change its power mode, it is recommended set the value as 2'b11. - if shared memory quadrant 2 does not map to CSM, set the value as 2'b11.
S3PM	[11:10]	0	Shared memory quadrant 3 power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if software does not want to change its power mode, it is recommended set the value as 2'b11. - if shared memory quadrant 3 does not map to CSM, set the value as 2'b11.

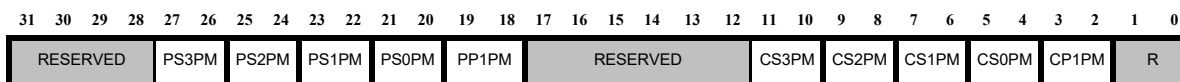
32.2.14 Partial Power-Down Cluster Shared Memory Register, PDM_CSM_P_PSTAT

Address: 0x628

Access: R

Default: based on CSM partition number configure and SM configuration register value

Figure 32-15 PDM_CSM_P_PSTAT Register



The PDM_CSM_P_PSTAT register indicates CSM partial power information, including the previous partial power status and current partial power status for each partition/quadrant. This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

For information about the power domains, see the *DesignWare ARCv2-based Processor Databook*.

Table 32-15 PDM_CSM_P_PSTAT Field Description

Field	Bit	Reset	Description
CP1PM	[3:2]	0	CSM partition 1 memory current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Note: if CSM partition number is 1, the field always returns 2'b11.
CS0PM	[5:4]	0	Shared memory quadrant 0 current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if shared memory quadrant 0 does not map to CSM, set the value as 2'b11.

Table 32-15 PDM_CSM_P_PSTAT Field Description

Field	Bit	Reset	Description
CS1PM	[7:6]	0	Shared memory quadrant 1 current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - If shared memory quadrant 1 does not map to CSM, the field always returns 2'b11.
CS2PM	[9:8]	0	Shared memory quadrant 2 current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if shared memory quadrant 2 does not map to CSM, the field always returns 2'b11.
CS3PM	[11:10]	0	Shared memory quadrant 3 current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if shared memory quadrant 3 does not map to CSM, the field always returns 2'b11.
PP1PM	[19:18]	0	CSM partition 1 previous memory power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Note: if CSM partition number is 1, the field always returns 2'b11.

Table 32-15 PDM_CSM_P_PSTAT Field Description

Field	Bit	Reset	Description
PS0PM	[21:20]	0	Shared memory quadrant 0 previous power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if shared memory quadrant 0 does not map to CSM, the field always returns 2'b11.
PS1PM	[23:22]	0	Shared memory quadrant 1 previous power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if shared memory quadrant 1 does not map to CSM, the field always returns 2'b11.
PS2PM	[25:24]	0	Shared memory quadrant 2 previous power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if shared memory quadrant 2 does not map to CSM, the field always returns 2'b11.
PS3PM	[27:26]	0	Shared memory quadrant 3 previous power mode⁰ 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if shared memory quadrant 3 does not map to CSM, the field always returns 2'b11.

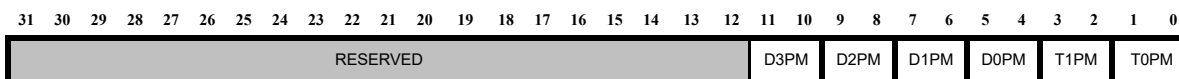
32.2.15 Partial Power-Down L2 Cache Register, PDM_SLC_P_PMODE

Address: 0x629

Access: RW

Default: 0x0000_0000

Figure 32-16 PDM_SLC_P_PMODE Register



Use this register to program the partial power-down mode of an L2 cache. This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

For information about the power domains, see the *DesignWare ARCV2-based Processor Databook*.

Table 32-16 PDM_SLC_P_PMODE Field Description

Field	Bit	Reset	Description
T0PM	[1:0]	0	L2 cache tag0 memory power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Note: if software does not want to change its power mode, it is recommended set the value as 2'b11.
T1PM	[3:2]	0	L2 cache partition 1 memory power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Note: if software does not want to change its power mode, it is recommended set the value as 2'b11.

Table 32-16 PDM_SLC_P_PMODE Field Description

Field	Bit	Reset	Description
D0PM	[5:4]	0	L2 cache data bank 0 power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if software does not want to change its power mode, it is recommended set the value as 2'b11.
D1PM	[7:6]	0	L2 cache data bank 1 power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if software does not want to change its power mode, it is recommended set the value as 2'b11.
D2PM	[9:8]	0	L2 cache data bank 2 power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if software does not want to change its power mode, it is recommended set the value as 2'b11.
D3PM	[11:10]	0	L2 cache data bank 3 power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value Notes: - if software does not want to change its power mode, it is recommended set the value as 2'b11.

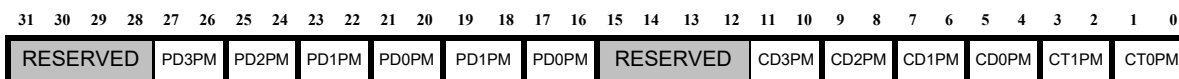
32.2.16 Partial Power-Down L2 Cache Register, PDM_SLC_P_PSTAT

Address: 0x62A

Access: R

Default: 0x0000_0000

Figure 32-17 PDM_SLC_P_PSTAT Register



The PDM_SLC_P_PSTAT register indicates the L2 cache partial power information, including the previous partial power status and current partial power status for each partition/quadrant. This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

For information about the power domains, see the *DesignWare ARCV2-based Processor Databook*.

Table 32-17 PDM_SLC_P_PSTAT Field Description

Field	Bit	Reset	Description
CT0PM	[1:0]	0	L2 cache tag0 memory current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value
CT1PM	[3:2]	0	L2 tag 1 memory current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value
CD0PM	[5:4]	0	L2 cache data bank 0 current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value
CD1PM	[7:6]	0	L2 cache data bank 1 current power mode 0x0: Power up 0x1: Power down, retention mode 0x2: Power down, shut down mode 0x3: Invalid value

Table 32-17 PDM_SLC_P_PSTAT Field Description

Field	Bit	Reset	Description
CD2PM	[9:8]	0	L2 cache data bank 2 current power mode ■ 0x0: Power up ■ 0x1: Power down, retention mode ■ 0x2: Power down, shut down mode ■ 0x3: Invalid value
CD3PM	[11:10]	0	L2 cache data bank 3 current power mode ■ 0x0: Power up ■ 0x1: Power down, retention mode ■ 0x2: Power down, shut down mode ■ 0x3: Invalid value
PT0PM	[17:16]	0	L2 cache tag0 memory previous power mode ■ 0x0: Power up ■ 0x1: Power down, retention mode ■ 0x2: Power down, shut down mode ■ 0x3: Invalid value
PT1PM	[19:18]	0	L2 tag1 memory previous power mode ■ 0x0: Power up ■ 0x1: Power down, retention mode ■ 0x2: Power down, shut down mode ■ 0x3: Invalid value
PD0PM	[21:20]	0	L2 cache data bank 0 previous power mode ■ 0x0: Power up ■ 0x1: Power down, retention mode ■ 0x2: Power down, shut down mode ■ 0x3: Invalid value ■
PD1PM	[23:22]	0	L2 cache data bank 1 previous power mode ■ 0x0: Power up ■ 0x1: Power down, retention mode ■ 0x2: Power down, shut down mode ■ 0x3: Invalid value

Table 32-17 PDM_SLC_P_PSTAT Field Description

Field	Bit	Reset	Description
PD2PM	[25:24]	0	L2 cache data bank 2 previous power mode ■ 0x0: Power up ■ 0x1: Power down, retention mode ■ 0x2: Power down, shut down mode ■ 0x3: Invalid value
PD3PM	[27:26]	0	L2 cache data bank 3 previous power mode ■ 0x0: Power up ■ 0x1: Power down, retention mode ■ 0x2: Power down, shut down mode ■ 0x3: Invalid value

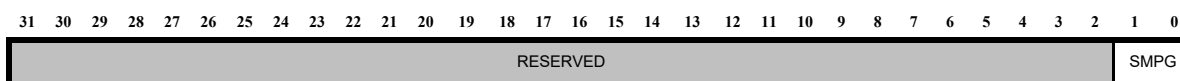
32.2.17 Shared Memory Power Down Register, PDM_CSM_SM_P_CFG

Address: 0x62B

Access: RW

Default: 0x0000_0000

Figure 32-18 PDM_CSM_SM_P_CFG Register



The PDM_CSM_SM_P_CFG register can be used to program the shared memory powered down granularity which is allocated to the CSM. This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

This register exists when the option CC_ALLOW_CSM_L2_SHARE_MEMORY is enabled.

If the shared memory is enabled and CSM partial power down feature is used, this register must be configured before using CSM.

Table 32-18 PDM_CSM_SM_P_CFG Field Description

Field	Bit	Reset	Description
SMPG	[1:0]	0	CSM allocated shared memory partial power down granularity 0x0: Power up 0x1: Partial power down granularity is 1/4 0x2: Partial power down granularity is 1/2 0x3: Partial power down granularity is 1/1

The SMPG field in PDM_CSM_SM_P_CFG is not cleared by hardware. The field retains the value programmed by the software.

The partial power granularity value can not be larger than the shared memory size that is allocated to the CSM. [Table 32-19](#) describes the legal values.

Table 32-19 Legal Combination value for SM_CFG and PDM_CSM_SM_P_CFG

Shared Memory Size for CSM	SMPG Partial Power Down granularity			
	0	1/4	1/2	1/1
1	✓	✓	✓	✓
1/2	✓	✓	✓	×

Table 32-19 Legal Combination value for SM_CFG and PDM_CSM_SM_P_CFG

Shared Memory Size for CSM	SMPG Partial Power Down granularity			
	0	1/4	1/2	1/1
1/4	✓	✓	×	×

After the SMPG field is configured, when the software programs the PDM_CSM_P_PMODE register to partially power down the shared memory, the partial power down size must be $\bar{N} * \text{SMPG}$ value. For example:

- If SM_SIZE == 1 and SMPG == 1/4, then software can program partial power down size is 1/4, 2/4, 3/4 and 4/4
- If SM_SIZE == 1 and SMPG == 1/2, then software can program partial power size is 1/2 and 2/2, 1/4 and 3/4 is forbidden.

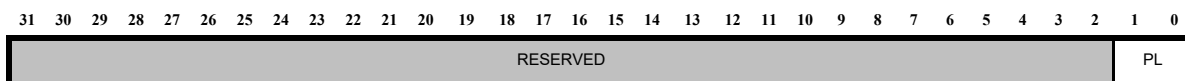
32.2.18 Cluster Performance Register, DVFS_CLUSTER_PL

Address: 0x626

Access: RW

Reset: 0x0000_0000

Figure 32-19 DVFS_CLUSTER_PL Register



Write to this register to set the performance level of a cluster. When you write to this register, a request is sent to the external DVFS controller to change the performance level accordingly. Reading this register returns the current performance level of the cluster. For more information, see Cluster Interface to the External DVFS in the Databook.

This register is present in the cluster top power always domain, and this register is shared by all the cores in a cluster.

Table 32-20 DVFS_CLUSTER_PL Field Description

Field	Bit	Reset	Description
PL	[1:0]	0	Current performance level of a cluster. 0x0: PL0 (performance level 0). This is the default value. 0x1: PL1 (performance level 1) 0x2: PL2 (performance level 2) 0x3: PL3 (performance level 3)

32.3 Execution Rate Interface

You can control the dynamic power (execution control) consumption of the core by limiting the maximum instruction execution rate of the core. Use the following auxiliary register to control the dynamic power consumption.

Table 32-21 Execution Control Registers

Address	Name	Description
0x08	Execution Rate Control Register, EXEC_CTRL	Execution rate control register

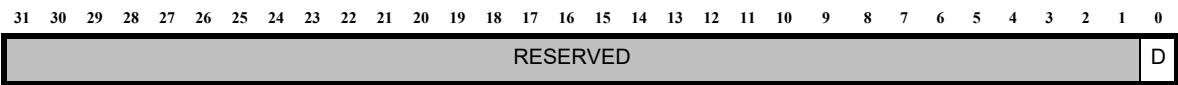
32.3.1 Execution Rate Control Register, EXEC_CTRL

Address: 0x08

Access: RW

Reset 0x0000_0000

Figure 32-20 EXEC_CTRL Register



The EXEC_CTRL register specifies whether the core can execute instructions in parallel. This register is present in the core only when the dual-issue feature is configured (MICRO_ARCH_BUILD[23:16]=0x04).

Table 32-22 EXEC_CTRL Field Description

Field	Bit	Reset	Description
D	[0]	0	Specifies if the core can execute instructions in parallel. 0x0: Core executes instructions in parallel at the highest rate achievable; that is dual-issue is enabled. 0x1: Core executes instructions one at a time; that is, the dual issue feature is disabled.

32.4 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.

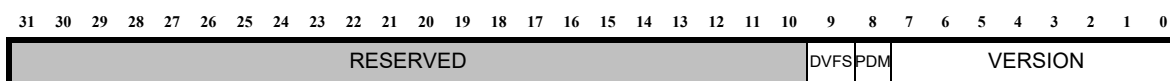
Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented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

32.4.1 Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD

Address: 0xF7

Access: R

Figure 32-21 PDM_DVFS_BUILD Register



This register indicates the presence of the power-domain management and DVFS capabilities in an ARCV2-based processor.

Table 32-23 PDM_DVFS_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	0x3: initial version ■ ■ 0x4: includes support for the following: - partial power up and power down of CSM and L2 caches - power management of shared memory configuration of L2 cache and CSM. - power optimizations for CSM, BIU, and L2 cache ■
PDM	[8]	Indicates whether the processor power consumption can be controlled by dividing the processor into different power domains. ■ 0x1: The processor can be divided into different power domains. ■ 0x0: The processor is not divided into different power domains.
DVFS	[9]	Indicates whether the DVFS (dynamic voltage and frequency scaling) functionality is supported. ■ 0x1: DVFS is supported. ■ 0x0: DVFS is not supported.

33

ARConnect

33.1 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.

33.2 ARConnect Register Interface

Table 33-1 ARConnect Registers

Address	Name	Description
0X600	CONNECT_CMD	ARConnect command register. Each core uses this register to send commands to the ARConnect. This register is present only when ARConnect is configured.
0x601	CONNECT_WDATA	Write data register. Each core uses this register to write data to ARConnect. This register is present only when ARConnect is configured.
0x602	CONNECT_READBACK	Read data register. Each core uses this register to read data from ARConnect. This register is present only when ARConnect is configured.
0xD0	CONNECT_SYSTEM_BUILD	Build configuration for: ARConnect System
0xD1	CONNECT_SEMAPHORE_BUILD	Build configuration for: ARConnect inter-core semaphore unit
0xD2	CONNECT_MESSAGE_BUILD	Build configuration for: ARConnect inter-core message unit
0xD3	CONNECT_PMU_BUILD	Build configuration for: ARConnect power management unit
0xD5	CONNECT_IDU_BUILD	Build configuration for: inter-core interrupt distribution unit

Table 33-1 (Continued)ARConnect Registers

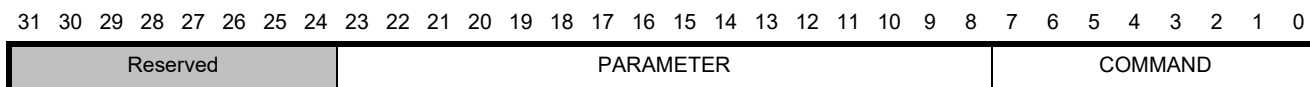
Address	Name	Description
0xD6	CONNECT_GFRC_BUILD	Build configuration for: ARConnect global free running counter
0xD7	CONNECT_IVC_BUILD	Build configuration for: ARConnect interrupt vector base control unit
0xE0	CONNECT_ICI_BUILD	Build configuration for: ARConnect inter-core interrupt unit
0xE1	CONNECT_ICD_BUILD	Build configuration for: ARConnect inter-core debug unit
0xE2	CONNECT_ASI_BUILD	Build configuration for: ARConnect slave interface unit
0xE3	CONNECT_PDM_BUILD	Build configuration for: ARConnect power domain management unit

33.2.1 ARConnect Command Register, CONNECT_CMD

Address: 0x600

Access: RW

Figure 33-1 CONNECT_CMD Register



The `CONNECT_CMD` register is used to send commands to ARConnect. For example, an ARC core can use the SR instruction to write specific commands to the `CONNECT_CMD` register, and the ARC core passes this command to ARConnect (to generate interrupts to other cores, or pass messages to other cores, and so on). The ARC core can use the LR instruction to read the `CONNECT_CMD` register to check the content of the last written command. In earlier releases, this register was named `MCIP_CMD`.

Table 33-2 CONNECT_CMD Register Field Description

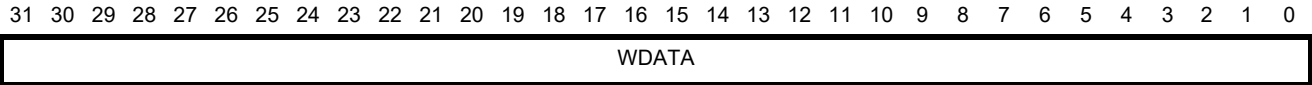
Field	Bits	Reset	Description
COMMAND	[7:0]	0	Command to ARConnect.
PARAMETER	[23:8]	0	Parameter field of the command to ARConnect. Any parameter that is required by a command is supplied in this field.
RESERVED	[31:24]	0	Reserved. Read-as-zero; ignored on write

33.2.2 ARConnect Write Data Register, CONNECT_WDATA

Address: 0x601

Access: RW

Figure 33-2 CONNECT_WDATA Register



The CONNECT_WDATA register is used to store data that should be passed from the ARC core to ARConnect. Some ARConnect commands use the data that you specify in this register.

The layout of the CONNECT_WDATA register depends on the ARConnect command being issued. In earlier releases, this register was named MCIP_WDATA.

Table 33-3 CONNECT_WDATA Register Field Description

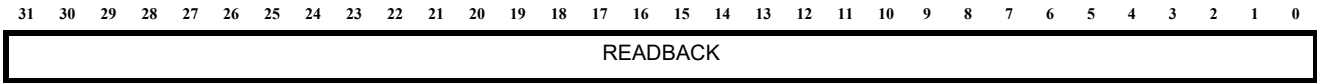
Field	Bits	Reset	Description
WDATA	[31:0]	0	The store data passed from ARC core to ARConnect

33.2.3 ARConnect Read Data Register, CONNECT_READBACK

Address: 0x602

Access: R

Figure 33-3 CONNECT_READBACK Register



The CONNECT_READBACK register returns the read back data when an ARC core issues any ARConnect-related read command. This register is read-only; writes to it are ignored and an illegal-instruction exception is raised.

The layout of the CONNECT_READBACK register depends on the issued ARConnect command. In earlier releases, this register was named MCIP_READBACK.

Table 33-4 CONNECT_READBACK Register Field Description

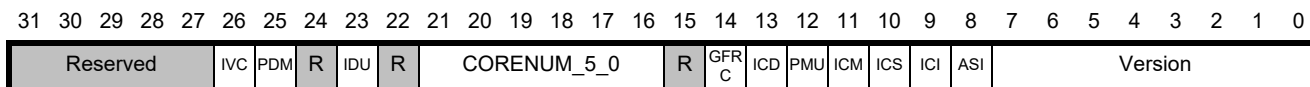
Field	Bits	Reset	Description
READBACK	[31:0]	0	The read back data when an ARC core issues any ARConnect related read command

33.2.4 ARConnect Build Configuration Register, CONNECT_SYSTEM_BUILD

Address: 0xD0

Access: R

Figure 33-4 CONNECT_SYSTEM_BUILD Register



The `CONNECT_SYSTEM_BUILD` register is an optional BCR register in the ARC cores. If the ARC core is not configured with ARConnect, reading this register returns zero. In earlier releases, this register was named as `MCIP_SYSTEM_BUILD`.

Table 33-5 CONNECT_SYSTEM_BUILD Register Field Description

Field	Bits	Description
VERSION	[7:0]	The ARConnect version. All non-enumerated values are reserved. 0x1: First version 0x2: Current version; - Changed the starting <code>CORE_ID</code> to 0 from 1 (first version). - Renamed the component from <code>MCIP</code> to <code>ARCONNECT</code>.
ASI	[8]	Presence of ARConnect Slave interface Unit 0x0: No ARConnect slave unit 0x1: Has ARConnect slave unit
ICI	[9]	Presence of Inter-Core Interrupt Unit 0x0: No inter-core interrupt unit 0x1: Has inter-core interrupt unit
ICS	[10]	Presence of Inter-Core Semaphore Unit 0x0: No inter-core semaphore unit 0x1: Has inter-core semaphore unit
ICM	[11]	Presence of Inter-Core Message Unit 0x0: No inter-core message unit 0x1: Has inter-core message unit
PMU	[12]	Presence of Power Management Unit 0x0: No power management unit 0x1: Has power management unit

Table 33-5 CONNECT_SYSTEM_BUILD Register Field Description

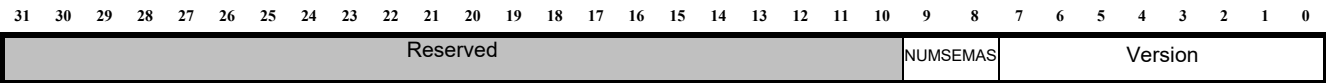
Field	Bits	Description
ICD	[13]	Presence of Inter-Core Debug Unit 0x0: No inter-core debug unit 0x1: Has inter-core debug unit
GFRC	[14]	Presence of Global Free-Running Counter Unit 0x0: No global free-running counter 0x1: Has global free-running counter
RESERVED	[15]	Reserved. Read-as-zero; ignored on write
CORENUM_5_0_	[21:16]	Number of cores (ARC and non-ARC) connected to ARConnect For example 6'h01: 1 core; 6'h02: 2 cores; 6'h04: 4 cores.
RESERVED	[22]	Reserved. Read-as-zero; ignored on write
IDU	[23]	Presence of Interrupt Distribution Unit 0x0: No interrupt distribution unit 0x1: Has interrupt distribution unit
RESERVED	[24]	Reserved. Read-as-zero; ignored on write
PDM	[25]	Presence of Power Domain Management Unit 0x0: No ARConnect power domain management unit 0x1: Has ARConnect power domain management unit
IVC	[26]	Presence of Interrupt Vector base Control Unit 0x0: No interrupt vector base control unit 0x1: Has interrupt vector control unit
RESERVED	[31:27]	Reserved. Read-as-zero; ignored on write

33.2.5 Inter-core Semaphore Unit BCR, CONNECT_SEMA_BUILD

Address: 0xD1

Access: R

Figure 33-5 CONNECT_SEMA_BUILD Register



The CONNECT_SEMA_BUILD register is an optional BCR register. If the Inter-Core Semaphore Unit is not configured in ARConnect, reading the CONNECT_SEMA_BUILD register returns zero. In earlier releases, this register was named MCIP_SEMA_BUILD.

Table 33-6 CONNECT_SEMA_BUILD Register Field Description

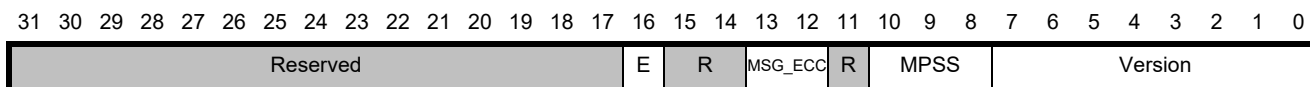
Field	Bits	Description
Version	[7:0]	Inter-core Semaphore Unit version. 0x1: Current version
NUMSEMAS	[9:8]	Number of Semaphores: 0x0: 8 semaphores 0x1: 16 semaphores 0x2: 32 semaphores 0x3: reserved
RESERVED	[31:10]	Reserved. Read-as-zero; ignored on write

33.2.6 Inter-core Message Unit BCR, CONNECT_MESSAGE_BUILD

Address: 0xD2

Access: R

Figure 33-6 CONNECT_MESSAGE_BUILD Register



The `CONNECT_MESSAGE_BUILD` register is an optional BCR register. If the Inter-Core Message Unit is not configured in ARConnect, reading the `CONNECT_MESSAGE_BUILD` register returns zero. In earlier releases, this register was named `MCIP_MESSAGE_BUILD`. This register exists only when `mcip_has_msg_sram==true` in your processor build.

Table 33-7 CONNECT_MESSAGE_BUILD Register Field Description

Field	Bits	Description
Version	[7:0]	Inter-core Message Unit version. 0x1: initial version. 0x2: Support for ECC protection on SRAM. 0x3: Added single-bit error counter for ECC.
MPSS	[10:8]	Size of the message-passing SRAM 0x0: 128 B 0x1: 256 B 0x2: 512 B 0x3: 1 KB 0x4: 2 KB 0x5: 4 KB - Other values are reserved
RESERVED	[11]	Reserved. Read-as-zero; ignored on write
MSG_ECC	[13:12]	ECC or parity protection scheme of Message SRAM 0x0: No protection 0x1: Parity protection 0x2: ECC (SECCDED) protection 0x3: ECC(SECCDED) with all 0/1 protection
RESERVED	[15:14]	Reserved. Read-as-zero; ignored on write

Table 33-7 CONNECT_MESSAGE_BUILD Register Field Description

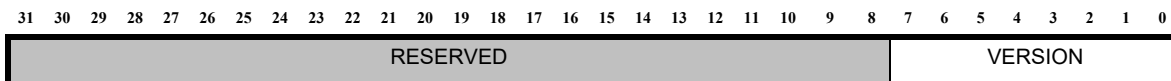
Field	Bits	Description
E (ECC_ADDR)	[16]	Specifies whether ECC protection is configured on message SRAM address ■ 0x0: No protection ■ 0x1: ECC protection is configured
RESERVED	[31:17]	■ Reserved. Read-as-zero; ignored on write

33.2.7 Power Management Unit BCR, CONNECT_PMU_BUILD

Address: 0xD3

Access: R

Figure 33-7 CONNECT_PMU_BUILD Register



The `CONNECT_PMU_BUILD` register is an optional BCR. If the PMU is not configured in ARConnect, reading the `CONNECT_PMU_BUILD` register returns zero. In earlier releases, this register was named `MCIP_PMU_BUILD`.

Table 33-8 CONNECT_PMU_BUILD Field Description

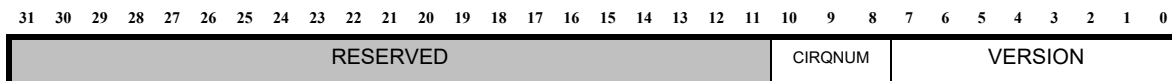
Field	Bit	Description
VERSION	[7:0]	Version number ■ 0x1: First version ■ 0x2: Second version; deleted the DVFS feature from ARConnect PMU; moved the power management feature to PDM (Power Domain Management); redefined PMU as optional external Power Management Unit. ■ 0x3: Current version; Added two additional counters (<code>RSTCNT</code> and <code>PDCNT</code>) and four relevant commands: <code>CMD_PMU_SET_RSTCNT</code>, <code>CMD_PMU_READ_RSTCNT</code>, <code>CMD_PMU_SET_PDCNT</code>, <code>CMD_PMU_READ_PDCNT</code>
RESERVED	[31:27]	■ Reserved. Read-as-zero; ignored on write

33.2.8 Interrupt Distribution Unit BCR, CONNECT_IDU_BUILD

Address: 0xD5

Access: R

Figure 33-8 CONNECT_IDU_BUILD Register



The `CONNECT_IDU_BUILD` register is an optional BCR. If the PMU is not configured in ARConnect, reading the `CONNECT_IDU_BUILD` register returns zero. In earlier releases, this register was named `MCIP_IDU_BUILD`.

Table 33-9 CONNECT_IDU_BUILD Field Description

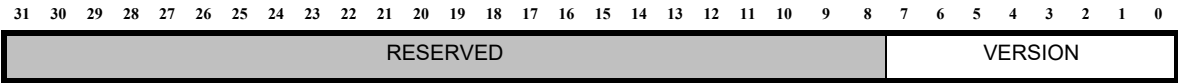
Field	Bit	Description
VERSION	[7:0]	Version number ■ 0x1: First version ■ 0x2: Current version; Added software acknowledge mode and support for single core.
CIRQNUM	[10:8]	Number of common interrupts supported in IDU: ■ 0x0 = 4 common interrupts ■ 0x1 = 8 common interrupts ■ 0x2 = 16 common interrupts ■ 0x3 = 32 common interrupts ■ 0x4 = 64 common interrupts ■ 0x5 = 128 common interrupts ■ Other values are reserved
RESERVED	[31:211]	■ Reserved. Read-as-zero; ignored on write

33.2.9 ARConnect Global Free Running Counter BCR, CONNECT_GFRC_BUILD

Address: 0xD6

Access: R

Figure 33-9 CONNECT_GFRC_BUILD Register



The CONNECT_GFRC_BUILD register is an optional BCR. If the global free running counter is not configured in ARConnect, reading the CONNECT_GFRC_BUILD register returns zero.

Table 33-10 CONNECT_GFRC_BUILD Field Description

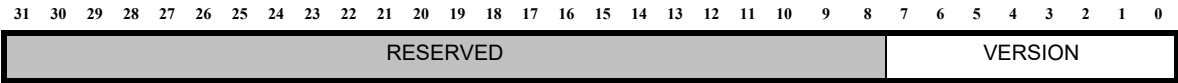
Field	Bit	Description
VERSION	[7:0]	Version number 0x1: First version 0x2: Second version; Renamed the global real-time counter (GRTC) to global free running counter (GFRC). 0x3: Third Version; Added the following commands: CMD_GFRC_SET_CORE, CMD_GFRC_READ_CORE, and CMD_GFRC_READ_HALT. 0x4: Current version; Added the following commands: CMD_GFRC_CLK_ENABLE, CMD_GFRC_CLK_DISABLE, CMD_GFRC_READ_CLK_STATUS
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write

33.2.10 Inter-Core Interrupt Vector Control Unit BCR, CONNECT_IVC_BUILD

Address: 0xD7

Access: R

Figure 33-10 CONNECT_IVC_BUILD Register



The CONNECT_IVC_BUILD register is an optional BCR that indicates support for setting the interrupt vector base using an external pin. If the interrupt vector base control unit is not configured in ARConnect, reading the CONNECT_IVC_BUILD register returns zero.

Table 33-11 CONNECT_IVC_BUILD Field Description

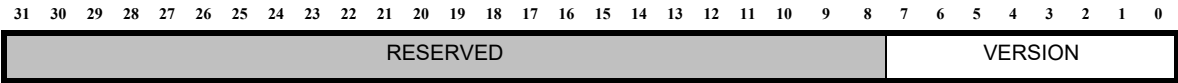
Field	Bit	Description
VERSION	[7:0]	Version number 0x1 : initial version. Indicates support for setting the interrupt vector base using an external pin.
RESERVED	[31:8]	■ Reserved. Read-as-zero; ignored on write

33.2.11 Inter-Core Interrupt Unit BCR, CONNECT_ICI_BUILD

Address: 0xE0

Access: R

Figure 33-11 CONNECT_ICI_BUILD Register



The CONNECT_ICI_BUILD register is an optional BCR. If the inter-core interrupt unit is not configured in ARConnect, reading the CONNECT_ICI_BUILD register returns zero.

Table 33-12 CONNECT_ICI_BUILD Field Description

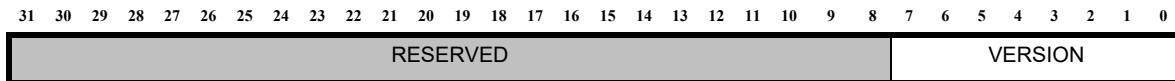
Field	Bit	Description
VERSION	[7:0]	Version number 0x2 : This register did not exist in earlier versions of ARConnect (version 0x1). 0x3: Current version; Added the following ICI command: CMD_INTRPT_GENERATE_ACK_BIT_MASK. 0x4: Current version; Added the following commands: CMD_INTRPT_EXT_MODE, CMD_INTRPT_SET_PULSE_CNT, CMD_INTRPT_READ_PULSE_CNT.
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write

33.2.12 Inter-Core Debug Unit BCR, CONNECT_ICD_BUILD

Address: 0xE1

Access: R

Figure 33-12 CONNECT_ICD_BUILD Register



The `CONNECT_ICD_BUILD` register is an optional BCR. If the inter-core debug unit is not configured in ARConnect, reading the `CONNECT_ICD_BUILD` register returns zero.

Table 33-13 CONNECT_ICD_BUILD Field Description

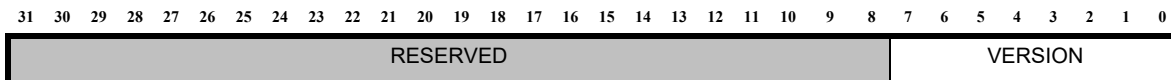
Field	Bit	Description
VERSION	[7:0]	Version number 0x2 : Current version. This register did not exist in earlier versions of ARConnect.
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write

33.2.13 ARConnect Slave Interface Unit BCR, CONNECT_ASI_BUILD

Address: 0xE2

Access: R

Figure 33-13 CONNECT_ASI_BUILD Register



The `CONNECT_ASI_BUILD` register is an optional BCR. If the ARConnect slave interface (ASI) unit is not configured in ARConnect, reading the `CONNECT_ASI_BUILD` register returns zero.

Table 33-14 CONNECT_ASI_BUILD Field Description

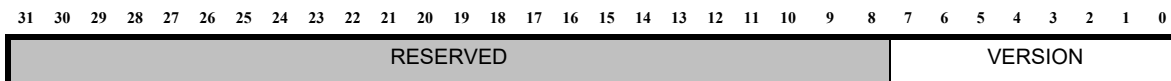
Field	Bit	Description
VERSION	[7:0]	Version number 0x2 : Current version; This register did not exist in earlier versions of ARConnect (version 0x1). ASI was called the Bus Slave Unit (BSU) in earlier version. This unit cannot access the low-latency memory. 0x3: Current version; Support for ECC on bus interface.
BUS_ECC_EN	[9:8]	0x0: No BUS-ECC protection 0x1: BUS-ECC protection without all 0/1 protection 0x2: BUS-ECC protection with all 0/1 protection 0x3: Reserved
BUS_ECC_PARITY_OPTION (PA)	[10]	0x0: EVEN parity protection for control signals in AXI buses 0x1: ODD parity protection for control signals in AXI buses
RESERVED	[31:10]	Reserved. Read-as-zero; ignored on write

33.2.14 ARConnect PDM BCR, CONNECT_PDM_BUILD

Address: 0xE3

Access: R

Figure 33-14 CONNECT_PDM_BUILD Register



The `CONNECT_PDM_BUILD` register is an optional BCR (exists when `mcip_power_domains==true`). If the power-domain management unit is not configured in ARConnect, reading the `CONNECT_PDM_BUILD` register returns zero.

Table 33-15 CONNECT_PDM_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number 0x1 : Current version
RESERVED	[31:8]	■ Reserved. Read-as-zero; ignored on write

33.3 Programming Restrictions

33.3.1 Atomic Operation

Operation sequences to the ARConnect are considered as atomic operations. The software must save and store the auxiliary registers if the atomic operation is interrupted.

For example: If CORE1 writes the PMU mode, the following instructions must be executed:

- (1) `SR 0x00000001, [CONNECT_WDATA];`
- (2) `SR 0x00000051, [CONNECT_CMD];`

If an interrupt occurs between these two instructions, the sequence is disrupted. The software must save the `CONNECT_WDATA` value, if it is written in the interrupt handler, and restore the value before it exits the interrupt.

33.3.2 Preventing Conflicts

Some ARConnect components such as the PMU require the software to guarantee that no simultaneous-access conflict happens. If two or more cores read or write these components simultaneously, the hardware result is unpredictable.

For example: If CORE1 sets the ICD mode for ICD, it uses the following operation sequence:

1. Read the ICD internal status register
2. Check the internal status

3. Update the ICD internal status register

Meanwhile, if CORE2 sets the IDU mode simultaneously, the operation sequences from two cores could be overlapped. Only one of them can be selected and performed, and the other sequence might be discarded. Therefore, software needs to guarantee no such access conflict happens.

Use the following methods to ensure that there are no simultaneous accesses:

- Utilize the ARConnect Inter-core Semaphores.
- Utilize the ARConnect Inter-core Message Unit to pass specific messages synchronizing different cores.
- Use specific address in the shared memory locked by LLOCK/SCOND as software semaphore if these two instructions are supported.

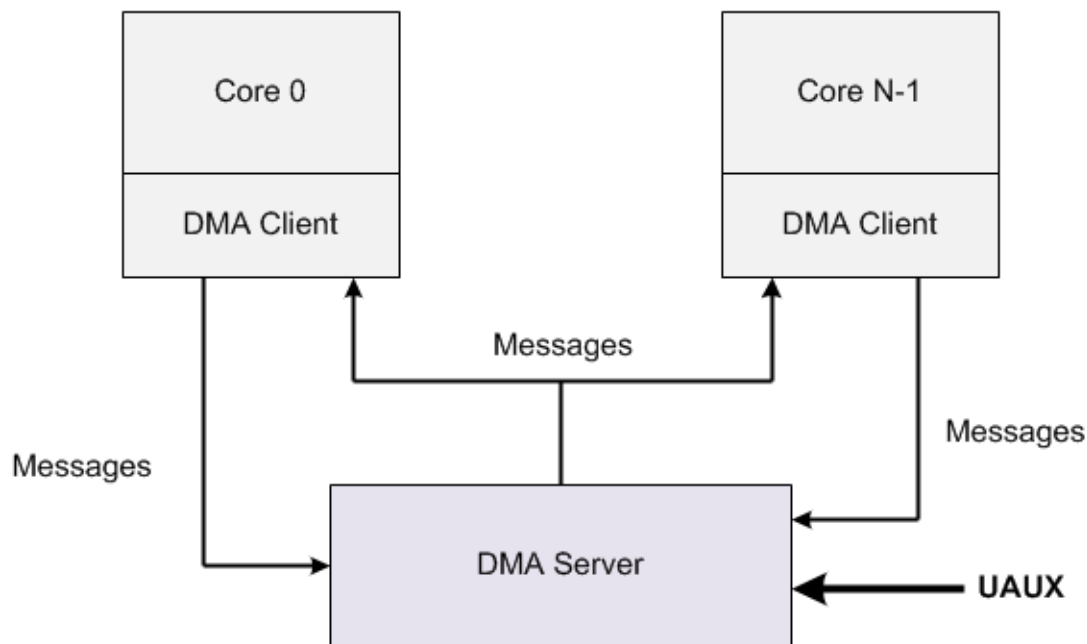
34

Cluster DMA Controller

34.1 High-Level Architecture

The cluster DMA controller uses a client-server architecture. Multiple DMA clients, one client for each core, share a single DMA server, see [Figure 34-1](#).

Figure 34-1 Cluster DMA Client – Server Architecture



Communication between the client and the server for setting up descriptors and handling responses is mostly through asynchronous (non-blocking) messages, so that the core is not stalled. DMA clients and the DMA server include functionality to:

- set up messages in the DMA-client auxiliary registers
- trigger message transfer from the client to the server
- receive messages from the server

- track the status of pending messages

The DMA client inside the processor cores includes support for polling-based, interrupt-based and event-based synchronization. A DMA client has a single interrupt at core level.

The DMA server is a component in Architect. If a DMA server is present, the corresponding DMA client components are instantiated implicitly at core level; a DMA server component can only be instantiated at the cluster level.

The cluster DMA controller uses descriptors, handles, channels, and bus transactions.

A descriptor defines an entire DMA transfer, copying data from one physical address range to another physical address range. A DMA descriptor includes at least four fields:

- the source physical byte address, from which the data needs to be copied
- the destination physical byte address, to which the data needs to be copied
- the length of the DMA transfer. The length is in the range [1B:128KB-1B] bytes and supports arbitrary byte alignments
- attributes of the descriptor. For example, a field to indicate if an interrupt needs to be raised when the DMA server has finished processing the descriptor. For more information about the attributes of the descriptor, see the `DMA_C_ATTR_AUX` register.

A descriptor is assigned to a channel in the DMA server. Each channel includes a FIFO for storing descriptors associated with that channel. A channel processes its descriptors in order. Descriptors in different channels may be processed out of order. A channel splits up large transfers into bursts for the system bus (AXI) or the internal buses. The burst length does not exceed the maximum burst length configured; these bursts are referred to as bus transactions.

A new DMA descriptor is pushed into a channel by copying the descriptor from the client into one of the DMA server channels. Upon successful pushing, the DMA server returns a handle to the DMA client, the handle being a pointer to the associated descriptor in the DMA server. The handle can be used to poll the status of the DMA transfer.

Channels with equal priority are arbitrated in a round-robin fashion. All channels have equal priorities.

The DMA server backend takes care of issuing and tracking the source bus read transactions and the associated destination bus write transactions.

A memory is used for storing all descriptors that have been pushed into a DMA channel. The memory is embedded in the DMA server; it is not accessible using processor load/store operations and is not memory-mapped for the processor cores.

Figure 34-2 Cluster DMA High-Level Architecture



- DMA client: core logic for issuing and tracking DMA transfers for each core.
- DMA server frontend: logic shared by all cores with channel logic for each channel.
- Descriptor memory-access logic: a wrapper around the single-port descriptor memory to run the memory at a fraction of the clock and arbitrate between multiple clients; logically the memory is part of the frontend.
- DMA server backend: logic performing bus transactions.

If a cluster includes a DMA server, each core includes a DMA client that includes local auxiliary registers and an interface for configuring the DMA server and polling status. DMA clients send atomic messages to the DMA server channel logic. Descriptors are stored in the descriptor memory. The arbiter selects the highest-priority channel that is allowed to packetize one bus transaction. The bus command packetizer reads the head of the channel-descriptor FIFO and generates a single bus transaction for the descriptor, updating and saving the descriptor in descriptor memory. The command issue module allocates a region in the data buffer and allocates an entry in the pending transaction table while issuing the bus-read command. When read data is returned it gets buffered and forwarded to one of the destination write interfaces. The pending-transaction table tracks all pending transactions. The response handler issues interrupts and updates channel state upon DMA transfer completion.

34.2 Use Cases

For the use case of the cluster DMA, see the *DesignWare ARC HS Series Databook*.

34.3 Configuration Options

For notes about the configuration options and their dependencies, see the *DesignWare ARC HS Series Databook*.

34.4 Programming the Cluster DMA using Auxiliary Registers

The cluster DMA controller auxiliary registers are categorized into the following:

- Build configuration register DMA_BUILD
- DMA client auxiliary registers, private to each core; these registers have the suffix DMA_C_
- DMA server auxiliary registers, shared among the cores; these registers have the suffix DMA_S_

The DMA auxiliary registers (except the build configuration register) start at auxiliary address DMA_AUX_BASE (=0xD00) and occupy a 256-address aperture. [Table 34-1](#) lists the DMA BCR and AUX registers. Addresses within the 256 register aperture not listed in the table are reserved and cannot be read or written.

Descriptor interrupt and status-related information is grouped for each 32 descriptors; these registers use suffix *D* to indicate the group. Channel-specific registers have a suffix *C* for the channel index.

All the auxiliary registers are non-serializing.

Table 34-1 Cluster DMA Auxiliary Register List

DMA_AUX_BASE+0x0	DMA Client Control Register, DMA_C_CTRL_AUX
DMA_AUX_BASE+0x1	DMA Client Channel Select Register, DMA_C_CHAN_AUX
DMA_AUX_BASE+0x2	DMA Client Source Address Register, DMA_C_SRC_AUX
DMA_AUX_BASE+0x3	DMA Client Source High Address Register, DMA_C_SRC_HI_AUX
DMA_AUX_BASE+0x4	DMA Client Destination Address Register, DMA_C_DST_AUX
DMA_AUX_BASE+0x5	DMA Client Destination High Address Register, DMA_C_DST_HI_AUX
DMA_AUX_BASE+0x6	DMA Client Attributes Register, DMA_C_ATTR_AUX

Table 34-1 Cluster DMA Auxiliary Register List

DMA_AUX_BASE+0x7	DMA Client Length Register, DMA_C_LEN_AUX
DMA_AUX_BASE+0x8	DMA Client Handle Register, DMA_C_HANDLE_AUX
DMA_AUX_BASE+0xA	DMA Event Status Register, DMA_C_EVSTAT_AUX
DMA_AUX_BASE+0xB	DMA Client Clear Event Status Register, DMA_C_EVSTAT_CLR_AUX
DMA_AUX_BASE+0xC	DMA Client Status Register, DMA_C_STATUS_AUX
DMA_AUX_BASE+0xD	DMA Client Interrupt Status Register, DMA_C_INTSTAT_AUX
DMA_AUX_BASE+0xE	DMA Client Clear Interrupt Status Register, DMA_C_INTSTAT_CLR_AUX
DMA_AUX_BASE+0xF	DMA Client Error Handle Register, DMA_C_ERRHANDLE_AUX
DMA_AUX_BASE+0x10	DMA Server Control Register, DMA_S_CTRL_AUX
DMA_AUX_BASE+0x20+D	DMA Server Done Status Register, DMA_S_DONESTATD_AUX
DMA_AUX_BASE+0x40+D	DMA Server Clear Done Status Register, DMA_S_DONESTATD_CLR_AUX
DMA_AUX_BASE+0x80+(C*8)	DMA Server Channel Tail Pointer Register, DMA_S_TAILC_AUX
DMA_AUX_BASE+0x81+(C*8)	DMA Server Channel Middle Pointer Register, DMA_S_MIDC_AUX
DMA_AUX_BASE+0x82+(C*8)	DMA Server Channel Head Pointer Register, DMA_S_HEADC_AUX
DMA_AUX_BASE+0x83+(C*8)	DMA Server Channel Base Register, DMA_S_BASEC_AUX
DMA_AUX_BASE+0x84+(C*8)	DMA Server Channel Last Register, DMA_S_LASTC_AUX
DMA_AUX_BASE+0x85+(C*8)	DMA Channel Priority Register, DMA_S_PRIORC_AUX
DMA_AUX_BASE+0x86+(C*8)	DMA Channel Control Register, DMA_S_STATC_AUX

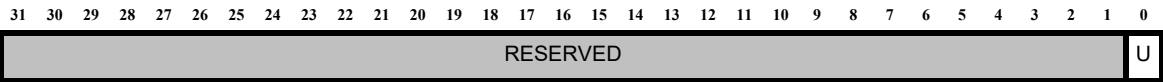
34.4.1 DMA Client Control Register, DMA_C_CTRL_AUX

Address: DMA_AUX_BASE+0x0

Access: RW

Reset: 0x0

Figure 34-3 DMA_C_CTRL_AUX Register



Use this register to control access to the DMA client registers in kernel mode and user mode.

Table 34-2 DMA_C_CTRL_AUX Field Description

Field	Bit	Description
U	[0]	0x1: The DMA client auxiliary registers are accessible in user and kernel modes. 0x0: The DMA client auxiliary registers are accessible in kernel mode only.

34.4.2 DMA Client Channel Select Register, DMA_C_CHAN_AUX

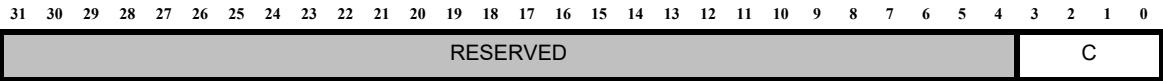
Address: DMA_AUX_BASE+0x1

Access:

- RW when DMAC_C_CTRL_AUX.U==0
- rw when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-4 DMA_C_CHAN_AUX Register



Use this register to select the target channel for a DMA descriptor, by specifying the channel index. If the DMA controller is configured for one channel only, the channel-index field has 0 bits, and register reads return zero, and writes are ignored.

The DMA_C_CHAN_AUX register must be written before new DMS descriptors are pushed (before writing the DMA_C_LEN_AUX register).

To guarantee atomic pushing of descriptors, auxiliary read and write accesses to this register are stalled as long as the DMA_C_STAT_AUX.B (busy) bit is set.

This register is part of the process context and must be saved and restored on a context switch.

Table 34-3 DMA_C_CHAN_AUX Field Description

Field	Bit	Description
C	[3:0]	Specifies the channel index. The width of this field depends on the maximum number of DMA channels (DMA_NUM_CHAN) configured in the processor: $C = \log_2(\text{DMA\_NUM\_CHAN})$

34.4.3 DMA Client Source Address Register, DMA_C_SRC_AUX

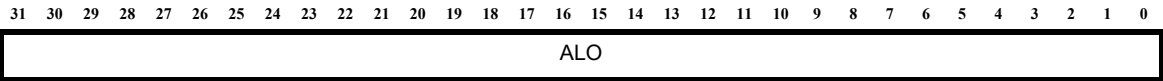
Address: DMA_AUX_BASE+0x2

Access:

- RW when DMAC_C_CTRL_AUX.U==0
- rw when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-5 DMA_C_SRC_AUX Register



Use this register to specify the source byte address for the DMA descriptor. When the physical-address extension (40-bit addresses) is configured, this register stores the lower 32 bits of the address.

The DMA_C_SRC_AUX register must be written before pushing a new descriptor, that is, before writing the DMA_C_LEN_AUX register.

To guarantee atomic pushing of descriptors, auxiliary read and write accesses to this register are stalled as long as the DMA_C_STAT_AUX.B (busy) bit is set.

This register is part of the process context and must be saved and restored on a context switch.

Table 34-4 DMA_C_SRC_AUX Field Description

Field	Bit	Description
ALO	[31:0]	DMA client source byte address for the DMA descriptor

34.4.4 DMA Client Source High Address Register, DMA_C_SRC_HI_AUX

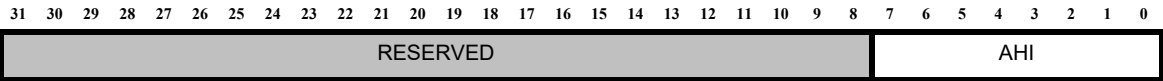
Address: DMA_AUX_BASE+0x3

Access:

- RW when DMAC_C_CTRL_AUX.U==0
- rw when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-6 DMA_C_SRC_HI_AUX Register



Use this register to specify the source byte address for the DMA descriptor. When the physical-address extension (40-bit addresses) is configured, this register stores the upper eight bits of the address. If the physical-address extension is not configured in the processor, this register does not exist.

The DMA_C_SRC_HI_AUX register must be written before new DMA descriptors are pushed, that is, before the DMA_C_LEN_AUX register is written.

To guarantee atomic pushing of descriptors, auxiliary read and write accesses to this register are stalled as long as the DMA_C_STAT_AUX.B (busy) bit is set.

This register is part of the process context and must be saved and restored on a context switch.

Table 34-5 DMA_C_SRC_HI_AUX Field Description

Field	Bit	Description
AHI	[7:0]	DMA client source byte address (upper 8 bits) for the DMA descriptor.

34.4.5 DMA Client Destination Address Register, DMA_C_DST_AUX

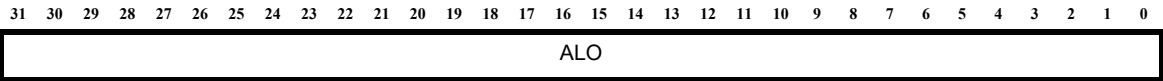
Address: DMA_AUX_BASE+0x4

Access:

- RW when DMAC_C_CTRL_AUX.U==0
- rw when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-7 DMA_C_DST_AUX Register



Use this register to specify the destination byte address for the DMA descriptor. When the physical-address extension (40-bit addresses) is configured, this register stores the lower 32 bits of the address.

The DMA_C_DST_AUX register must be written before a new descriptor is pushed, that is, before the DMA_C_LEN_AUX register is written.

To guarantee atomic pushing of descriptors, auxiliary read and write accesses to this register are stalled as long as the DMA_C_STAT_AUX.B (busy) bit is set.

This register is part of the process context and must be saved and restored on a context switch.

Table 34-6 DMA_C_DST_AUX Field Description

Field	Bit	Description
ALO	[31:0]	DMA client destination byte address for the DMA descriptor

34.4.6 DMA Client Destination High Address Register, DMA_C_DST_HI_AUX

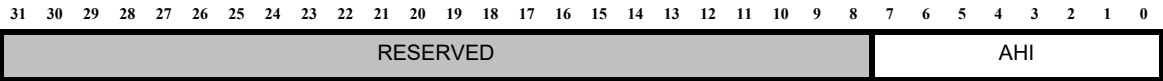
Address: DMA_AUX_BASE+0x5

Access:

- RW when DMAC_C_CTRL_AUX.U==0
- rw when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-8 DMA_C_DST_HI_AUX Register



Use this register to specify the destination byte address for the DMA descriptor. When the physical address extension (40-bit addresses) is configured, this register stores the upper eight bits of the address. If the physical-address extension is not configured in the processor, this register does not exist.

The DMA_C_DST_HI_AUX register must be written before a new descriptor is pushed, that is, before the DMA_C_LEN_AUX register is written.

To guarantee atomic pushing of descriptors, auxiliary read and write accesses to this register are stalled as long as the DMA_C_STAT_AUX.B (busy) bit is set.

This register is part of the process context and must be saved and restored on a context switch.

Table 34-7 DMA_C_DST_HI_AUX Field Description

Field	Bit	Description
AHI	[7:0]	DMA client source byte address (upper 8 bits) for the DMA descriptor.

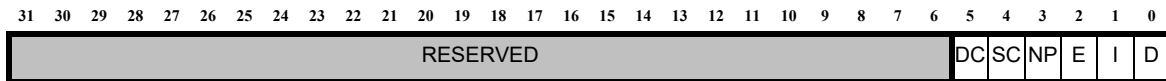
34.4.7 DMA Client Attributes Register, DMA_C_ATTR_AUX

Address: DMA_AUX_BASE+0x6

Access:
 ■ W when DMAC_C_CTRL_AUX.U==0
 ■ w when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-9 DMA_C_ATTR_AUX Register



Use this register to specify the DMA transfer attributes. This register is reset automatically after the DMA client pushes a DMA descriptor to the DMA server.

To guarantee atomic pushing of descriptors, auxiliary read and write accesses to this register are stalled as long as the DMA_C_STAT_AUX.B (busy) bit is set.

This register is part of the process context and must be saved and restored on a context switch.

Table 34-8 DMA_C_ATTR_AUX Field Description

Field	Bit	Description
D	[0]	Setting this bit to 1 sets the DMA_S_DONESTATD_AUX bit corresponding to the descriptor without raising an interrupt when the DMA server is done processing the descriptor. Setting this bit to 1 implies that polling is also enabled (DMA_C_ATTR_AUX.D==1).
I	[1]	Setting the DMA_C_ATTR_AUX.I bit triggers an interrupt and sets the DMA_S_DONESTAT bit corresponding to the descriptor when the DMA server is done processing the descriptor. Setting this bit to 1 implies that polling is also enabled (DMA_C_ATTR_AUX.D==1).
E	[2]	Set this bit to 1 to enable event-based synchronization mode. This mode of operation is safe as long as multiple processes on a single core access different channels only. Processes on different cores can still access the same DMA channel. To wake up, a core issues a new DMA descriptor while setting the EVENT attribute in the descriptor, then issues a wait-for-event instruction (WEVT). At DMA completion the server triggers a wake-up event in the originating core. Setting this bit to 1 implies that polling is also enabled (DMA_C_ATTR_AUX.D==1).

Table 34-8 DMA_C_ATTR_AUX Field Description

Field	Bit	Description
NP	[3]	Non-posted write enable. Set this bit to 1 to enable non-posted writes, that is, bufferable writes, on the bus for this descriptor. The value of this bit controls the value of the AXI <code>awcache[0]</code> output signal. If this bit is cleared, all bus transactions associated with a descriptor are non-posted, except for the last bus transaction, which is posted; if this bit is set, all transactions are posted.
SC	[4]	Source memory region coherent. This bit only exists when the CDMA is configured with the System Coherency Unit (SCU) and DMA coherency checking is enabled. This bit defines whether accesses to the CSM or external memory should be snooped first before each transaction, that makes up a block transfer, takes place; in other words when the source memory is coherent. 1 : memory snooping is requested. 0 : memory snooping is not requested.
DC	[5]	Destination memory region coherent. This bit only exists when the CDMA is configured with the System Coherency Unit (SCU) and DMA coherency checking is enabled. This bit defines whether accesses to the CSM or external memory should be snooped first before each transaction, that makes up a block transfer, takes place; in other words when the destination memory is coherent. 1 : memory snooping is requested. 0 : memory snooping is not requested.

**Note**

The event mode and the interrupt mode are mutually exclusive. Interrupt mode has priority, that is, writing a value {E,I, D} = 7 is read back as {E,I,D} = 3.

34.4.7.1 Coherent DMA transfers

Coherent transactions are split into bursts that are a maximum of data cache line length, overriding the setting of `DMA_S_CTRL_AUX.MLEN`. AXI transactions to coherent memory are also automatically truncated (if not aligned) against the cache line address boundary first and then subsequently transactions aligned to the cache line boundary.

If the `DMA_S_CTRL_AUX.MLEN` field is redefined by the software application to make the maximum burst length less than the cache line length, then all transactions that are not to coherent memory use this burst length, otherwise the burst length is set to data cache line length.

34.4.8 DMA Client Length Register, DMA_C_LEN_AUX

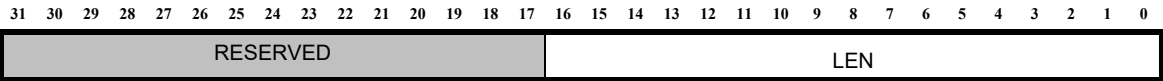
Address: DMA_AUX_BASE+0x7

Access:

- RW when DMAC_C_CTRL_AUX.U==0
- rw when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-10 DMA_C_LEN_AUX Register



Use this register to specify the length of the DMA transfer in bytes. The maximum DMA transfer size is 2^{LEN} bytes. A write to this register pushes the descriptor to the DMA server. Pushing the descriptor is an atomic operation and asynchronous. While the descriptor is pushed, the DMA_C_STAT_AUX.B (busy) bit is set. Any subsequent accesses to the auxiliary descriptor and handle registers are stalled until the DMA server acknowledges the push operation.

This register is part of the process context. However, this register does not need to be saved and restored on a context switch, as restoring this register triggers a new push event.

Table 34-9 DMA_C_LEN_AUX Field Description

Field	Bit	Description
LEN	[16:0]	Specifies the length of a DMA transfer.

34.4.9 DMA Client Handle Register, DMA_C_HANDLE_AUX

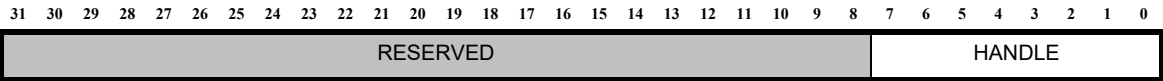
Address: DMA_AUX_BASE+0x8

Access:

- RW when DMAC_C_CTRL_AUX.U==0
- rw when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-11 DMA_C_HANDLE_AUX Register



This register contains the handle, the descriptor index, of the most recently pushed DMA descriptor. This register is updated asynchronously after the DMA server acknowledges a dma_push message.

To guarantee the integrity of this register, auxiliary read and write accesses to this register are stalled as long as the DMA_C_STAT_AUX.B (busy) bit is set, that is, while a dma_push message is pending.

Table 34-10 DMA_C_HANDLE_AUX Field Description

Field	Bit	Description
HANDLE	[7:0]	Specifies the descriptor index of the most recently pushed DMA descriptor.

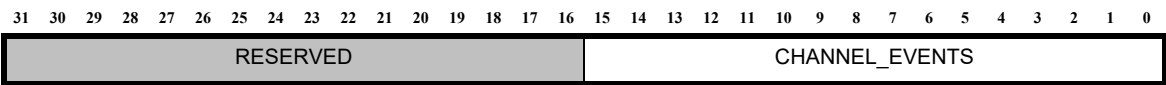
34.4.10 DMA Event Status Register, DMA_C_EVSTAT_AUX

Address: DMA_AUX_BASE+0xA

Access: ■ R

Reset: 0x0

Figure 34-12 DMA_C_EVSTAT_AUX Register



This register stores the DMA status for the core. This register has one bit for each channel.

To clear a bit in this register, write 1 to the corresponding bit in the DMA_C_EVSTAT_CLR_AUX register.

Table 34-11 DMA_C_EVSTAT_AUX Field Description

Field	Bit	Description
CHANNEL_EVENTS	[15:0]	A DMA event bit is set to 1 if all data for a descriptor has arrived at its final destination and if the descriptor DMA_C_ATTR_AUX.E bit was set when the descriptor was pushed to the server.

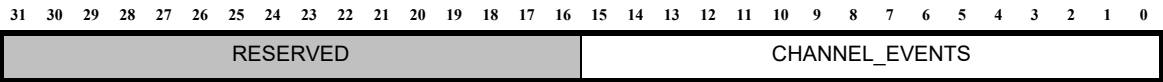
34.4.11 DMA Client Clear Event Status Register, DMA_C_EVSTAT_CLR_AUX

Address: DMA_AUX_BASE+0xB

Access: ■ W

Reset: 0x0

Figure 34-13 DMA_C_EVSTAT_CLR_AUX Register



Use this register to clear the DMA channel event status for the core.

Table 34-12 DMA_C_EVSTAT_CLR_AUX Field Description

Field	Bit	Description
CHANNEL_EVENTS	[15:0]	Set a DMA event bit to 1 to clear the corresponding bit in the DMA_C_EVSTAT_AUX register.

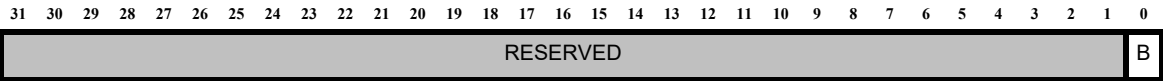
34.4.12 DMA Client Status Register, DMA_C_STATUS_AUX

Address: DMA_AUX_BASE+0xC

Access: ■ R when DMAC_C_CTRL_AUX.U==0
 ■ r when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 34-14 DMA_C_STATUS_AUX Register



This register stores the status of the asynchronous dma_push message. When you write to the DMA_C_LEN_AUX register, a dma_push message is triggered, and the DMAC_C_STATUS_AUX [0] bit is set. The DMAC_C_STATUS_AUX [0] bit is cleared when the dma_push message is acknowledged.

While the DMAC_C_STATUS_AUX [0] bit is set to 1, all auxiliary accesses to the descriptor-related registers are stalled.

Table 34-13 DMA_C_STATUS_AUX Field Description

Field	Bit	Description
B	[0]	■ 0x1: Indicates that a dma_push message is triggered. ■ 0x0: Indicates that the dma_push message is acknowledged.

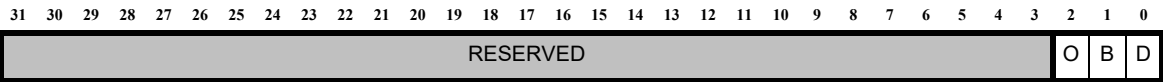
34.4.13 DMA Client Interrupt Status Register, DMA_C_INTSTAT_AUX

Address: DMA_AUX_BASE+0xD

Access: R

Reset: 0x0

Figure 34-15 DMA_C_INTSTAT_AUX Register



This register stores the interrupt status of a core. To clear a bit in this register, write 1 to the corresponding bit in the DMA_C_INTSTAT_CLR_AUX register.

A DMA client raises an interrupt to its corresponding core if:

- All write data for a descriptor has arrived at its final destination and the `int_en` attribute was set while pushing the descriptor. The originating DMA client of the descriptor sets its DMA_C_INTSTAT_AUX.D interrupt status bit.
- The DMA server received a read or write bus error response. The originating DMA client of the descriptor sets its DMA_C_INTSTAT_AUX.B interrupt status bit.
- A channel is full while a DMA client tries to push a new descriptor and the DMA_S_CTRL_AUX.O bit is set. The originating DMA client sets its DMA_C_INTSTAT_AUX.O interrupt status bit.

 **Note** The DMA interrupt signal to the core is the OR of all three interrupt bits.

Table 34-14 DMA_C_INTSTAT_AUX Field Description

Field	Bit	Description
D	[0]	■ 0x1: indicates that the data for a descriptor is transferred to the destination. This bit is set only if the DMA_C_ATTR_AUX.I bit for this descriptor is set when pushing the descriptor to the server. ■ 0x0: transfer is not done.

Table 34-14 DMA_C_INTSTAT_AUX Field Description

Field	Bit	Description
B	[1]	■ 0x1: Indicates that a descriptor in the channel has received a bus error response. When a bus error occurs, the DMA client does the following: - finish the bus transaction of the erroneous descriptor - abort further processing of the erroneous descriptor - optionally send a done interrupt and optionally set the done bit of the corresponding descriptor - set to 1 the DMA_C_INTSTAT_AUX.B bit in the originating client - set to 1 the erroneous descriptor handle value in the DMA_C_ERRHANDLE_AUX.HANDLE field ■ 0x0: Indicates that no bus error occurred.
O	[2]	■ 0x1: Channel has overflowed. ■ 0x0: Channel has not overflowed.

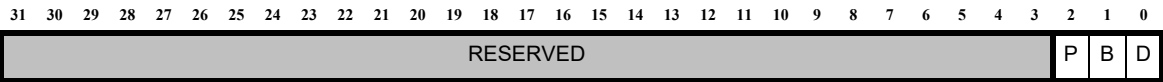
34.4.14 DMA Client Clear Interrupt Status Register, DMA_C_INTSTAT_CLR_AUX

Address: DMA_AUX_BASE+0xE

Access: W

Reset: 0x0

Figure 34-16 DMA_C_INTSTAT_CLR_AUX Register



Use this register to clear the DMA interrupt status for the core.

Table 34-15 DMA_C_INTSTAT_CLR_AUX Field Description

Field	Bit	Description
D	[0]	Write 1 to clear the interrupt-done bit in the DMA_C_INTSTAT_AUX register.
B	[1]	Write 1 to clear the bus-error bit in the DMA_C_INTSTAT_AUX register.
P	[2]	Write 1 to clear the overflow bit in the DMA_C_INTSTAT_AUX register.

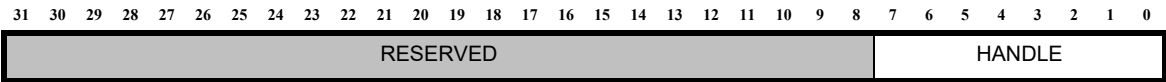
34.4.15 DMA Client Error Handle Register, DMA_C_ERRHANDLE_AUX

Address: DMA_AUX_BASE+0xF

Access: ■ R

Reset: 0x0

Figure 34-17 DMA_C_ERRHANDLE_AUX Register



This register contains the handle, the descriptor index, that caused the most recent bus error.

Table 34-16 DMA_C_ERRHANDLE_AUX Field Description

Field	Bit	Description
HANDLE	[7:0]	Specifies the descriptor index that caused the most recent bus error.

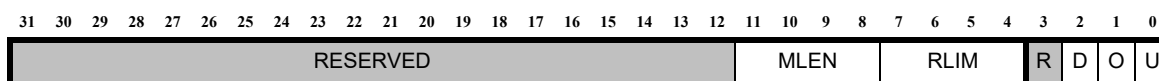
34.4.16 DMA Server Control Register, DMA_S_CTRL_AUX

Address: DMA_AUX_BASE+0x10

Access: ■ RW

Reset: 0x0

Figure 34-18 DMA_S_CTRL_AUX Register



This register controls the pseudo-static parameters for the DMA server.

Table 34-17 DMA_S_CTRL_AUX Field Description

Field	Bit	Description
U	[0]	User-mode access enable. 0x1: The DMA server auxiliary registers are accessible in user and kernel modes. 0x0: The DMA server auxiliary registers are accessible in kernel mode only.
O	[1]	Overflow error enable 0x1: The DMA server raises an interrupt in the DMA client if the client pushes a descriptor in a full channel. 0x0: The DMA server returns a retry acknowledgment if the DMA client pushes a descriptor in a full channel.
D	[2]	Default bus port for DMA transactions that do not map to any address space. 0x1: the LBU port is default; the IOC port is only used for CSM accesses. 0x0: the IOC port is default; the LBU port is only used for transactions explicitly hitting the LBU aperture
RLIM	[7:4]	LBU read transaction limit The DMA_S_CTRL_AUX.RLIM field specifies the maximum number of read bus transactions on the LBU port.
MLEN	[11:8]	Burst length $0 \leq \text{MLEN} \leq \log_2(\text{DMA\_MAX\_BURST})$ Specifies the maximum bus burst length as a $\log_2$. Effective maximum bus burst length is $\min(2^{\text{DMA\_S\_CTRL\_AUX.MLEN}}, \text{DMA\_SRV\_MAX\_BURST})$

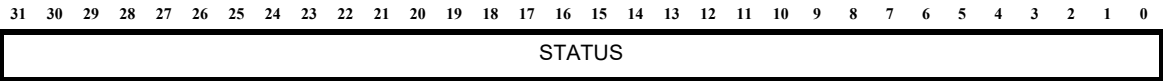
34.4.17 DMA Server Done Status Register, DMA_S_DONESTATD_AUX

Address: DMA_AUX_BASE+0x20+D

Access: ■ R when DMAC_S_CTRL_AUX.U==0
 ■ r when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 34-19 DMA_S_DONESTATD_AUX Register



The DMA_S_DONESTATD_AUX register stores the done status bits for the descriptors: $D*32$ through $D*32+31$.

If the descriptor DMA_C_ATTR_AUX.D or DMA_C_ATTR_AUX.i bits were set while the descriptor was written, the DMA server sets the DMA_S_DONESTATD_AUX.STATUS[B] bit corresponding to the descriptor handle. For a given handle H , the register index D and the bit offset B can be computed as:

$D = H/32$
 $B = H\%32$

A bit in the register can be cleared by writing a 1 in the corresponding bit of the DMA_S_DONESTATD_CLR_AUX register.

Table 34-18 DMA_S_DONESTATD_AUX Field Description

Field	Bit	Description
STATUS	[31:0]	Stores the done status bits for descriptors: $D*32$ through $D*32+31$

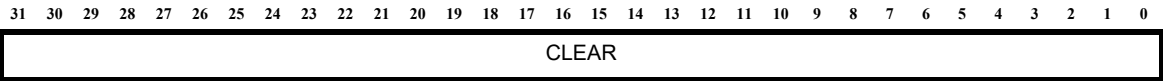
34.4.18 DMA Server Clear Done Status Register, DMA_S_DONESTATD_CLR_AUX

Address: DMA_AUX_BASE+0x40+D

Access: ■ W when DMAC_S_CTRL_AUX.U==0
 ■ w when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 34-20 DMA_S_DONESTATD_AUX Register



The DMA_S_DONESTATD_CLR_AUX register is used to clear the status bits (DMA_S_DONESTATD_AUX) for the descriptors: $D*32$ through $D*32+31$.

For a given handle H , the register index D and the bit offset B can be computed as:

$D = H/32$
 $B = H\%32$

Table 34-19 DMA_S_DONESTATD_AUX Field Description

Field	Bit	Description
CLEAR	[31:0]	Clear the status bits (DMA_S_DONESTATD_AUX) for the descriptors: $D*32$ through $D*32+31$.

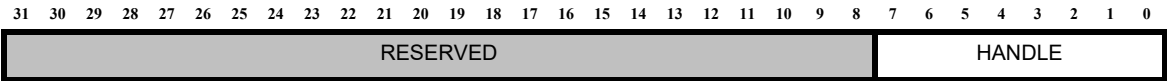
34.4.19 DMA Server Channel Tail Pointer Register, DMA_S_TAILC_AUX

Address: DMA_AUX_BASE+0x80+(C*8)

Access: ■ R when DMAC_S_CTRL_AUX.U==0
 ■ r when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 34-21 DMA_S_TAILC_AUX Register



The DMA_S_TAILC_AUX register points at the tail of the descriptor list of channel C. The register defines the handle value of the handle to be returned on the next dma_push. After a dma_push, the value of the register is incremented by one; if the register value matches DMA_S_LASTC_AUX, the next value is DMA_S_BASEC_AUX, effectively wrapping back. The register is (re)initialized to the value of the channel base pointer when the DMA_S_BASEC_AUX register is written to ensure the tail is pointing inside the descriptor region allocated to the channel.

This register is used for debug purposes only.

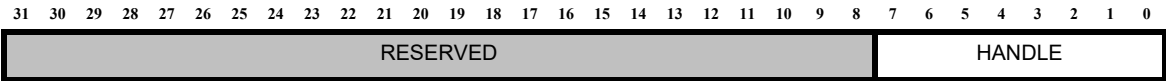
34.4.20 DMA Server Channel Middle Pointer Register, DMA_S_MIDC_AUX

Address: DMA_AUX_BASE+0x81+(C*8)

Access: ■ R when DMAC_S_CTRL_AUX.U==0
 ■ r when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 34-22 DMA_S_MIDC_AUX Register



The DMA_S_MIDC_AUX register points at the head of the descriptor list of channel C. The register defines the handle value of the next handle to be issued on the bus. After all bus transactions are issued, the value of the register is incremented by one; if the register value matches DMA_S_LASTC_AUX, the next value is DMA_S_BASEC_AUX, effectively wrapping back. The register is (re)initialized to the value of the channel base pointer when DMA_S_BASEC_AUX register is written to ensure the mid pointer is pointing inside the descriptor region allocated to the channel.

This register is used for debug purposes only.

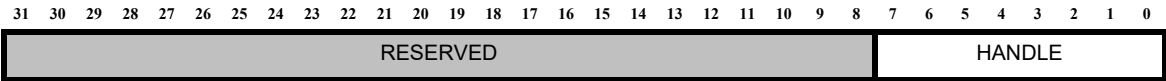
34.4.21 DMA Server Channel Head Pointer Register, DMA_S_HEADC_AUX

Address: DMA_AUX_BASE+0x82+(C*8)

Access: ■ R when DMAC_S_CTRL_AUX.U==0
 ■ r when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 34-23 DMA_S_HEADC_AUX Register



The DMA_S_HEADC_AUX register points at the head of the descriptor list of channel C. The register defines the handle value of the oldest handle that is non-idle, that is, the handle has a pending DMA transfer or a set DMA_S_DONESTAT bit. The register is incremented by one when the DMA completes and the head descriptor state has returned to IDLE. The value of the register is then incremented by one; if the register value matches DMA_S_LASTC_AUX, the next value is DMA_S_BASEC_AUX, effectively wrapping back. The register is (re)initialized to the value of the channel base pointer when DMA_S_BASEC_AUX register is written to ensure the head pointer is pointing inside the descriptor region allocated to the channel.

This register is used for debug purposes only.

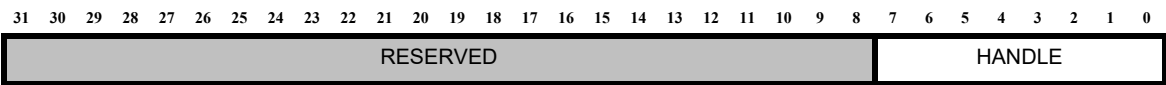
34.4.22 DMA Server Channel Base Register, DMA_S_BASEC_AUX

Address: DMA_AUX_BASE+0x83+(C*8)

Access: ▪ RW

Reset: 0x0

Figure 34-24 DMA_S_BASEC_AUX Register



The DMA_S_BASEC_AUX register defines the start of the descriptor region associated with channel C. The register value must be modified only when the DMA channel is disabled and empty. The value of the register must not be modified after the DMA_S_STATC_AUX.E bit is set to enable the channel. Writing a new value while the DMA channel is enabled or non-empty and results in undefined behavior.

The descriptor region for channel C as defined by the range [DMA_S_BASEC_AUX : DMA_S_LASTC_AUX] must not overlap other channel-descriptor ranges unless the overlapping channel is disabled. If channels with overlapping descriptor ranges are simultaneously enabled, the behavior is undefined. The same is true if channels with overlapping descriptor ranges are temporarily disabled and reenabled without reset.

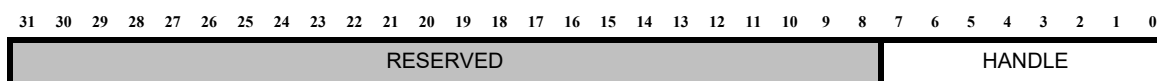
34.4.23 DMA Server Channel Last Register, DMA_S_LASTC_AUX

Address: DMA_AUX_BASE+0x84+(C*8)

Access: ■ RW

Reset: 0x0

Figure 34-25 DMA_S_LASTC_AUX Register



The DMA_S_LASTC_AUX register defines the end of the descriptor region associated with channel C. The register value may be modified only when the DMA channel is disabled and empty. The value of the register must not be modified after the channel is enabled by setting the DMA_S_STATC_AUX.E bit. Writing a new value while the DMA channel is enabled or non-empty results in undefined behavior.

The descriptor region for channel C as defined by the range [DMA_S_BASEC_AUX : DMA_S_LASTC_AUX] must not overlap other channel descriptor ranges unless the overlapping channel is disabled. If channels with overlapping descriptor ranges are simultaneously enabled, the behavior is undefined. The same is true if channels with overlapping descriptor ranges are temporarily disabled and reenabled without reset.

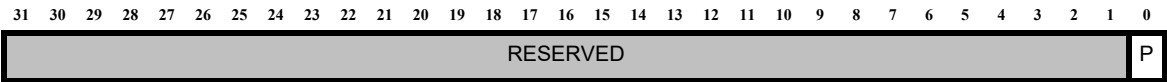
34.4.24 DMA Channel Priority Register, DMA_S_PRIORC_AUX

Address: DMA_AUX_BASE+0x85+(C*8)

Access: ■ RW

Reset: 0x0; all channels have same priority

Figure 34-26 DMA_S_PRIORC_AUX Register



The DMA_S_PRIORC_AUX register defines the priority for channel C. If DMA_S_PRIORC_AUX.P is set, the channel is high priority; if cleared then low priority. Channels with equal priorities are arbitrated using a round-robin scheme.

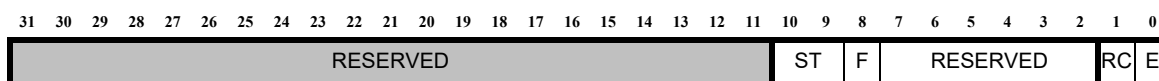
34.4.25 DMA Channel Control Register, DMA_S_STATC_AUX

Address: DMA_AUX_BASE+0x86+(C*8)

Access: ■ RW

Reset: 0x0; all channels have same priority

Figure 34-27 DMA_S_STATC_AUX Register



The DMA_S_STATC_AUX register defines the controls and status for channel C. The register includes controls for enabling and resetting the channel; the status fields include the channel state and full state.

Table 34-20 DMA_S_STATC_AUX Field Description

Field	Bit	Description
E	[0]	Enable Channel If DMA_S_STATC_AUX.E is set, the channel is enabled; clearing the bit disables the channel after the pending bus transactions are complete.
RC	[1]	Reset Channel If DMA_S_STATC_AUX.RC is set, the channel is reset; this bit is auto-cleared once the reset completes and the channel is in the IDLE state.
F	[8]	Channel Full State This bit is read-only, ignored on write.
ST	[10:9]	Channel State: This field reflects the channel FSM state. 0x0: idle 0x1: busy 0x2: stopped 0x3: reset This bit is read-only, ignored on write.

34.5 Steps to Initiate a DMA Transfer

To initiate a DMA transfer, do the following:

1. Write to DMA_S_CTRL_AUX to define the global attribute for DMA transfers.
2. Allocate the descriptor queue range for the respective channel by writing to DMA_S_BASEC_AUX and DMA_S_LASTC_AUX.

3. Enable the respective channel by writing to DMA_S_STATC_AUX.
4. Select the channel DMA_C_CHAN_AUX to process the next DMA descriptor.
5. Define the source and destination addresses for the transfer by writing to DMA_C_SRC_AUX and DMA_C_DST_AUX.
6. Set up the attributes for the transfer by writing to the DMA_C_ATTR_AUX.
7. Define the block size of the transfer by writing to DMA_C_LEN_AUX. The write initiates a push of the descriptor to the DMA server for processing.

Following the successful push of the descriptor to the DMA server, a value called the handle is returned in DMA_C_HANDLE_AUX from which the program can access status information related to the descriptor. Following is an example code for the set up DMA transfers.

```

; -----
; Issue DMA transfer branch-&-link function
; -----
; Arguments:
;
; r0 = channel number
; r1 = source address in bytes
; r2 = destination address in bytes
; r3 = transfer block size in bytes
; r4 = attribute for the transfer (that is, register DMA_C_ATTR_AUX)
;
; result:
; r0 = handle of transfer

issue_dma:
; first we locate the base register of server registers for this channel

    mov    r5, 0xD00                ; first we locate the base register of server
    lsr    r6, r0, 1                ; registers for this channel
    add    r6, 0x8, r6
    lsl    r6, r6, 4
    btst   r0, 0
    beq    1f
    add    r6, r6, 8
1:
    add    r5, r5, r6                ; located base register of server registers

; find the stat register
    add    r6, r5, 0x6                ; just enable the channel, bit 0 set to 1
    sr     1, [r6]

;---- Setting up Client Based registers ----
    sr     r0, [DMA_C_CHAN_AUX]        ; select the DMA channel for the descriptor
    sr     r1, [DMA_C_SRC_AUX]         ; select the source address
    sr     r2, [DMA_C_DST_AUX]         ; select the destination address
    sr     r4, [DMA_C_ATTR_AUX]        ; set the attributes for transfer
    sr     r3, [DMA_C_LEN_AUX]         ; triggers a DMA push to the server

; get the handle (stalls if push operation is busy)

```

```

    lr    r0, [DMA_C_HANDLE_AUX]    ; get the handle for the descriptor
    jal   [blink]                  ; retrain from function

```

With access to the handle the application can now poll the done bit in the correct DMA_S_DONESTATD register. The following code provides an example of executing wait for descriptor done function.

```

; -----
; Poll for DMA descriptor done branch-&-link function
; -----
; Arguments:
; r0 = handle

wait_dma_poll:
    mov    r1, DMA_AUX_BASE+0x020    ; get the correct DMA_S_DONESTATD register
    lsr    r2, r0, 5
    add    r2, r1, r2
    and    r3, r0, 0x1f
    lsl    r0, 1, r3
poll_done:
                                ; poll the correct done bit
    lr     r4, [r2]
    and    r4, r4, r0
    brne   r4, r0, poll_done
    jal    [blink]                ; exit function, the descriptor has been processed

; -----
; Clear DMA done branch-&-link function
; -----
; Arguments:
; r0 = handle

clr_dma_done:
    mov    r1, DMA_AUX_BASE+0x040    ; get the correct DMA_S_DONESTATD_CLR register
    lsr    r2, r0, 5
    add    r2, r1, r2
    and    r3, r0, 0x1f
    lsl    r0, 1, r3
    sr     r0, [r2]                ; clear the done bit
    jal    [blink]

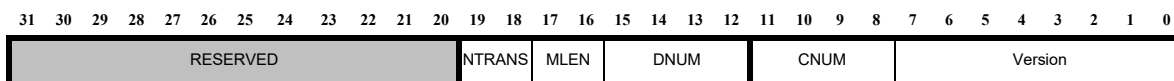
```

34.5.1 DMA Build Configuration Register, DMA_BUILD

Address: 0x0E6

Access: ■ R

Figure 34-28 DMA_BUILD Register



The DMA_BUILD register contains the version and configuration information of the Cluster DMA.

Table 34-21 DMA_BUILD Field Description

Field	Bit	Description
Version	[7:0]	■ 0x1: Initial version
CNUM	[11:8]	Specifies the number of channels in the DMA server $\log_2(\text{DMA\_SRV\_NUM\_CHAN})$
DNUM	[15:12]	Specifies the number of descriptors in the DMA server $\log_2(\text{DMA\_SRV\_NUM\_DESCR})$
MLEN	[17:16]	Specifies the maximum burst size for a bus transaction $\log_2(\text{DMA\_SRV\_MAX\_BURST}/4)$
NTRANS	[19:18]	Specifies the maximum number of pending bus transactions $\log_2(\text{DMA\_SRV\_MAX\_TRANS}/4)$

35

Floating-Point Unit

The floating-point unit consists of a set of hardware accelerated floating-point instructions compliant with IEEE 754-2008 specification.

35.1 Core Register Set

Figure 35-1 shows a summary of the core register set.

Figure 35-1 Core Register Map Summary

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
r0	Basecase Core Registers																															
r1																																
-																																
r24																																
r25																																
r26	Global Pointer (GP)																															
r27	Frame Pointer (FP)																															
r28	Stack Pointer (SP)																															
r29	Interrupt Link Register (ILINK)																															
r30	General Purpose Register																															
r31	Branch Link Register (BLINK)																															
r32																																
-	Extension Core Registers																															
r55r57																																
r56	Vector Stack Pointer																															
r57	Vector Frame pointer																															
r58	Accumulator Low ACCL (litle endian), ACCH (big-endian)																															
r59	Accumulator High- ACCH (litle endian), ACCL (big endian)																															

Figure 35-1 Core Register Map Summary (Continued)

r60	LP_COUNT[LPC_SIZE-1:0]	
r61	RESERVED	
r62	Long Immediate Data Indicator	
r63	Program Counter [PC_SIZE-1:2], Read-Only (PCL)	0 0

35.2 Extension Core Registers

The register set is extensible in register positions r32 to r57.

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).

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”.

When a floating-point unit is included and the `FSMADD`, `FSMSUB`, `FDMADD`, and `FDMSUB` instructions are enabled, the two extension core registers (r58 and r59) are included and comprise the third operand as follows:

Figure 35-2 ACCH for Double-Precision FPU operations

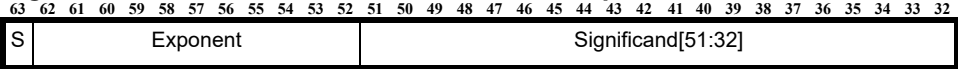


Figure 35-3 ACCL for Double-Precision FPU operations

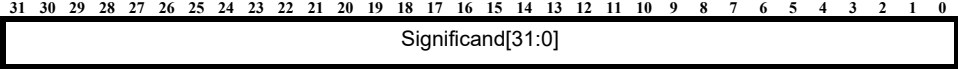
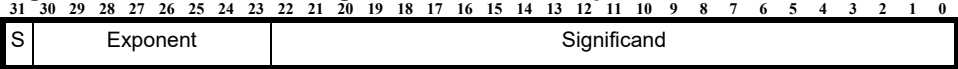


Figure 35-4 ACCL for Single-Precision FPU Operations



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.

35.3 FPU Auxiliary Registers

Table 35-1 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

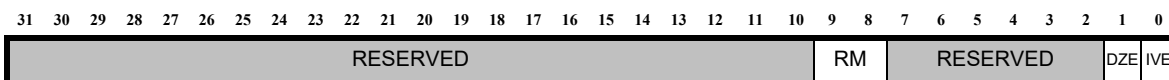
35.3.1 Floating-Point Unit Control Register, FPU_CTRL

Address: 0x300

Access: rw

Default 0x00000100

Figure 35-5 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 35-2 FPU_CTRL Field Description

Field	Bits	Reset	Description
IVE	0	0	Enable Invalid Operation Exception 0x0: Exception is not raised when an invalid operation occurs and the IV flag in the FPU_STATUS register is set. 0x1: Exception is raised when an invalid operation occurs and the IV flag in the FPU_STATUS register is not set.
DZE	1	0	Enable Divide by Zero Exception 0x0: Exception is not raised when a divide by zero operation occurs and the DZ flag in the FPU_STATUS register is set. 0x1: Exception is raised when a divide by zero operation occurs and the DZ flag in the FPU_STATUS register is not set. This exception is raised only by the hardware divider implementation. A software divider implementation does not raise this exception.
RESERVED	[7:2]	0	Reserved. Read-as-zero; ignored on write.

Table 35-2 FPU_CTRL Field Description

Field	Bits	Reset	Description
RM	[9:8]	1	Indicates Rounding Mode ■ 0x0: Round towards zero; the precise result is rounded to the nearest representable number closest to zero. ■ 0x1: Round to nearest even (default); the precise result is rounded to the nearest representable number. If the precise result is exactly between two representable numbers, the result is rounded to the even representable number. ■ 0x2: Round to positive infinity; the result is rounded to the next most positive representable number. ■ 0x3: Round to negative infinity; the result is rounded to the next most negative representable number.
RESERVED	[31:10]	0	Reserved. Read-as-zero; ignored on write.

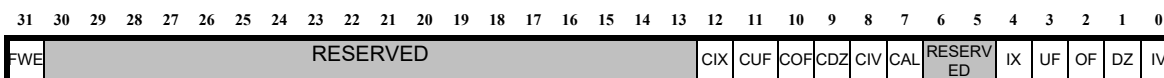
35.3.2 Floating-Point Unit Status Register, FPU_STATUS

Address: 0x301

Access: rw

Reset 0x00000000

Figure 35-6 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 35-3 FPU_STATUS Field Description

Field	Bits	Reset	Description
IV	0	0	Invalid operation sticky flag. This flag is updated implicitly based on the result of a floating-point operation. This flag is a sticky flag. You need to explicitly write to the CIV bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the CIV bit to clear this flag. ■ If FWE==1, this bit has both read and write access.
DZ	1	0	Divide-by-zero sticky flag. This flag is updated implicitly based on the result of a floating-point operation. This flag is a sticky flag. You need to explicitly write to the CDZ bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the CDZ bit to clear this flag. ■ If FWE==1, this bit has both read and write access.
OF	2	0	Overflow sticky flag. This flag is updated implicitly based on the result of a floating-point operation. This flag is a sticky flag. You need to explicitly write to the COF bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the COF bit to clear this flag. ■ If FWE==1, this bit has both read and write access.

Table 35-3 FPU_STATUS Field Description

Field	Bits	Reset	Description
UF	3	0	Underflow sticky flag. The underflow is checked and the underflow flag is updated before the result of the floating-point instruction is rounded (<i>before rounding</i> as declared in the std IEEE Std 754-2008). You need to explicitly write to the CUF bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the CUF bit to clear this flag. ■ If FWE==1, this bit has both read and write access.
IX	4	0	Inexact sticky flag. This flag is updated implicitly based on the result of a floating-point operation. This flag is a sticky flag. You need to explicitly write to the CIX bit to clear this flag. ■ If FWE==0, this bit is read only. Write 1 to the CIX bit to clear this flag. ■ If FWE==1, this bit has both read and write access.
RESERVED	[6:5]	0	Reserved. Read-as-zero; ignored on write.
CAL	7	0	Clear all flags. ■ If FWE==0, you can write to this bit. Write 1 to clear bits [4:0]. ■ If FWE==1, this bit read as zero and ignored on write.
CIV	8	0	Clear invalid operation flag. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the IV bit. ■ If FWE==1, this bit read as zero and ignored on write.
CDZ	9	0	Clear divide by zero. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the DZ bit. ■ If FWE==1, this bit read as zero and ignored on write.
COF	10	0	Clear overflow flag. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the OF bit. ■ If FWE==1, this bit read as zero and ignored on write.
CUF	11	0	Clear underflow flag. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the UF bit. ■ If FWE==1, this bit read as zero and ignored on write.
CIX	12	0	Clear inexact flag. ■ If FWE==0, you can write to this bit. Write 1 to this bit to clear the IX bit. ■ If FWE==1, this bit read as zero and ignored on write.

Table 35-3 FPU_STATUS Field Description

Field	Bits	Reset	Description
RESERVED	[30:11]	0	Reserved. Read-as-zero; ignored on write.
FWE	31	0	Flag Write Enable 0x0 : Enables write only access to bits [12:7]; bits[4:0] are read-only and writes are ignored; other bits are ignored on writes and read as zero. 0x1 : Enable write access to bits [4:0], these bits can be read-only if FWE==0. Other bits are read as zero and ignored on write.

IV – Invalid Operation

The invalid operation flag is set in the following cases:

- If any of the input operands to a floating-point instruction is a signaling NaN when `FPU_CTRL.IVE==0` (invalid operation exceptions is disabled).
- For `FCVT32`: Set when a floating-point number is converted to an integer, where the floating-point number is NaN, or infinity, or the conversion result is too big to be represented as an integer.
- For `FCVT32_64`: Set 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.
- For `FCVT64`: 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.
- For `FCVT64_32`: 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.
- 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.
- For `FDADD`: 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 `FDCMP`: Set when either of the inputs is a signaling NaN and the comparison result is unordered.

- For FDCMPF: Set when either of the inputs is a quiet or signaling NaN and the result is unordered.
- For FDDIV: Set for division operations of 0/0 and infinity/(infinity); or when an input operand is a signaling NaN.
- For FDMUL, FDMADD, FDMSUB: Set when multiplication of 0 x (infinity) or (infinity) x 0 occurs.; or when an input operand is a signaling NaN.
- For FDSQRT: Set when the c operand is less than -zero; or when an input operand is a signaling NaN.
- For FDSUB: Set when the operands are both +infinity or both are -infinity; or when an input operand is a signaling NaN.

DZ – Divide-by-Zero

The divide-by-zero flag is set when the divisor is zero (0), and the dividend is a finite non-zero number.

OF – Overflow

Overflow occurs when the result is too large to be represented by the instruction data format. The result is rounded and then saturated.

UF – Underflow

Underflow occurs when the result is too small to be represented by the instruction data format. The underflow flag is set before rounding. The result is rounded and then saturated.

IX – Inexact

Inexact operation occurs when the result of a floating-point operation is rounded and not exact.

35.4 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.

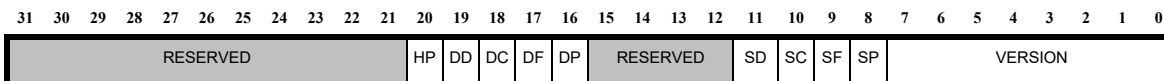
Auxiliary registers, in the range 0x60 to 0x7F and 0xC0 to 0xFF, are assumed to be BCRs. In kernel mode, any read from a non-existent build configuration register in this range returns 0, and no exception is generated. This design enables the kernel-mode code to detect the presence or absence of a BCR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

35.4.1 Floating-Point Unit Build Register, FPU_BUILD

Address: 0xC8

Access: R

Figure 35-7 FPU_BUILD Register



This register is present only if a floating-point unit is included in the processor configuration. [Table 35-4](#) lists the field descriptions:

Table 35-4 FPU_BUILD Field Descriptions

Field	Bits	Description
VERSION	[7:0]	Floating-Point Unit Version. All non-enumerated values are reserved. 0x01: First version of floating-point unit in ARCV2-based processors. With this version, support for single-precision is always true and also includes support for half-precision conversion instructions.
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.
RESERVED	[15:12]	Reserved. Read-as-zero; ignored on write.
DP	16	Indicates support for double-precision floating-point instructions
DF	17	Indicates support for double-precision fusedMultiplyAdd and fusedMultiplySubtract instructions.
DC	18	Indicates support for double-precision floating-point format conversion instructions.
DD	19	Indicates support for double-precision square-root and divide instructions.
HP	20	Indicates that the half-precision floating-point instructions are configured, that is, you can convert between half-precision and single-precision formats.

Table 35-4 FPU_BUILD Field Descriptions

Field	Bits	Description
RESERVED	[31:21]	Reserved. Read-as-zero; ignored on write.

36

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”](#).

37

ROM Patching Unit

37.1 Introduction

The ROM Patching Unit (RPU) allows you to patch a ROM incrementally. The RPU contains a programmable table of Patch Entries. When the core attempts to execute an instruction at an address that exactly matches the address of a valid patch entry, RPU triggers either a Patch Breakpoint or a Patch Jump, depending on whether the Patch Entry is programmed as a trap or a patch. [Figure 37-1](#) illustrates the RPU functionality. [Figure 37-2](#) illustrates the components of the ROM Patching Unit.

Figure 37-1 RPU Functionality

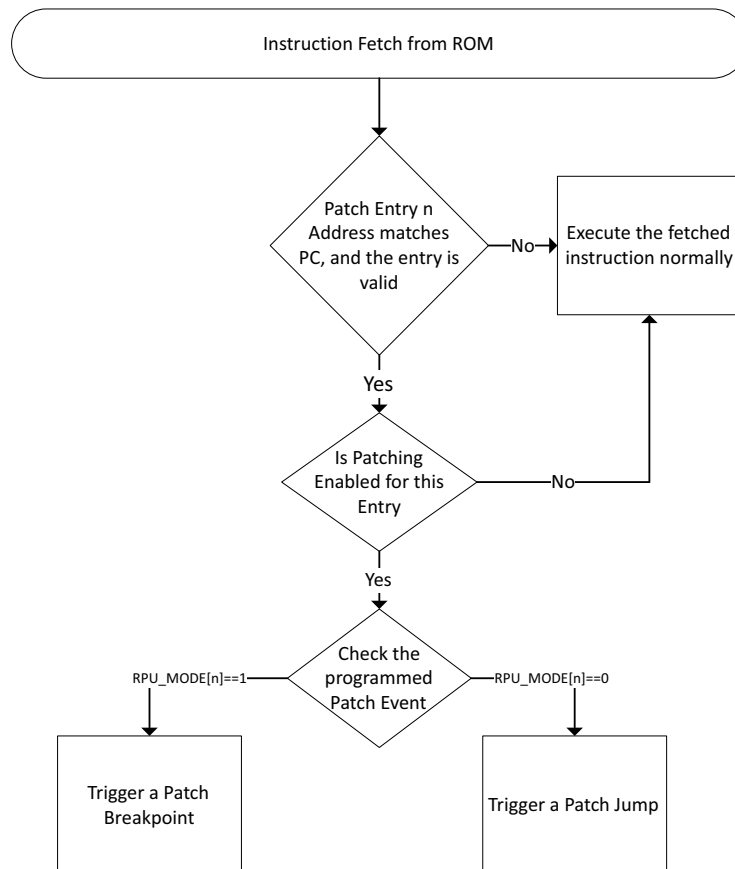
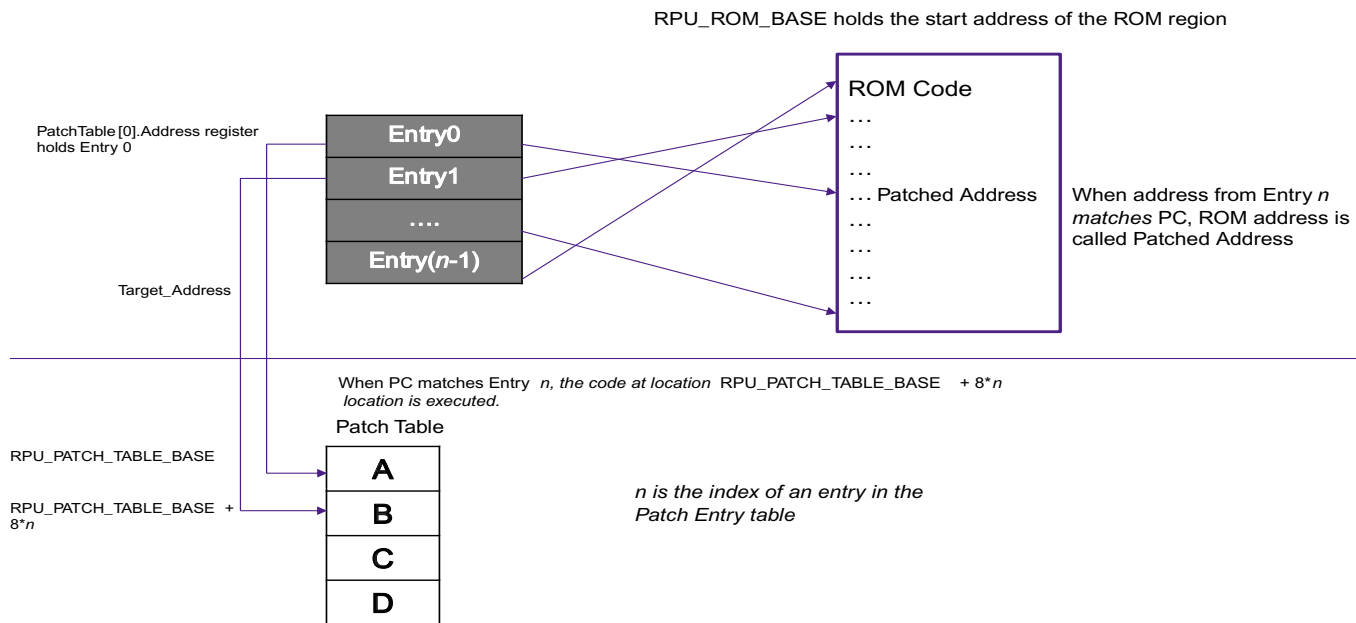


Figure 37-2 RPU Mapping

37.2 Patch Entry

Each Patch Entry is defined as a source address and control to decide on the action if a hit is detected. A hit occurs when the core attempts to execute an instruction at an address that exactly matches the address of a valid patch entry. Such instruction is called a patched instruction and such source address are called patched addresses. Each source address is defined as an offset from the start of the ROM region. All patch addresses are aligned to half-word (2-byte) locations, as all instructions are similarly aligned in the ARCV2 architecture.

When a hit occurs, RPU triggers either a Patch Breakpoint or a Patch Jump, depending on whether the Patch Entry is programmed as a trap or a patch. If any Patch Entry triggers a Patch Breakpoint, then no Patch Jumps are triggered for the source address.

37.3 Patch Breakpoints

When a Patch Breakpoint is taken, the processor halts with the PC pointing to the instruction that matched the patch address, just as if a BRK instruction had been placed at that location in ROM.

These breakpoints halt the core so that the debugger can take control. In case of a standalone execution without a debugger, the core remains halted until it is either reset or an external run request is issued.

37.4 Patch Jump

When a Patch Jump is taken, the processor effectively converts the current instruction to an unconditional Jump to the Target address:

$\text{Target\_address} = \text{base\_of\_patch\_table} + 8 * \text{patch\_entry\_number}$

This causes the PC to be set to the Target address of the triggered Patch Entry. The Patch Table is a table of target addresses. Each target address is 64-bit aligned. If patch entry n is triggered, and the patch event is Patch Jump, then PC is set to the target address of the n th entry. The patch table is accessed with an instruction fetch. The memory target depends on the address of the patch table in memory and the memory map of the system. The cacheable state of that reference depends on the cacheable state of the memory containing the patch table.

37.4.1 BPU

Processors that support branch prediction treat the implicit jump to the target address as a predictable unconditional branch (that is, branch type = BR_UNCONDITIONAL). This design means that the branch prediction unit (BPU) learns about the implicit jump and subsequently predicts that the instruction to fetch after the source address is the instruction from the target address. Once the BPU learns about the implicit jump, the additional cycle cost of executing a patch is one cycle.

If a patch is disabled, or a patch target address changes, the normal mis-prediction mechanism of the BPU results in the prediction being removed or updated.

37.4.2 Delay Slots

The parent branch of the delay slot must be patched rather than the delay slot itself. The resulting patch contains two instructions; the original branch (expanded to a jump if the branch target is now too far away) and the patched delay slot instruction.

37.5 Exceptions

If a fetch-related exception is raised when fetching an instruction at an address that triggers a Patch Event, or if an enabled interrupt is asserted, the fetch-related exception or interrupt is taken with higher priority than a Patch Event. The exception to this behavior is when the MPU detects instruction fetch protection violations; in this case, the patch event takes priority. In such scenarios, the workaround is to disable any ROM patching entry if its patch address does not have at least one of user or kernel execute permissions enabled in the MPU.

This design allows orthogonal mechanisms such as MMU, MPU or Actionpoints to continue to operate on addresses that have been patched. However, the conversion of the patched instruction into a Patch Event means that none of the instruction-related exceptions are taken. For example, if the patched instruction was an Illegal Instruction, no Illegal Instruction exception is raised once the instruction is patched.

If there is no fetch-related exception or interrupt, then the triggered Patch Event is taken.

37.6 Build-Time Decisions

You must decide on the following at the processor build time.

- Size of the ROM to be patched (`-rpu_rom_size`)
- Number of patch entries (`-rpu_patch_entries`)

37.7 Programming the RPU

1. Program the ROM base address.
 - a. Write `0x3` to the RPU_ADDR register.

- b. Write the ROM base address to the RPU_DATA register.
 - c. Write 0x1 to the RPU_CMND register.
2. Program the patch table base address.
 - a. Write 0x4 to the RPU_ADDR register.
 - b. Write the patch table base address to the RPU_DATA register.
 - c. Write 0x1 to the RPU_CMND register.
3. Write the ROM addresses that you want to patch. Repeat the sub-steps for each patch entry. The number of entries is defined during build configuration.
 - a. Write the index (0x10 to 0x1F) of the entry to the RPU_ADDR register.
 - b. Write the ROM offset address that you want to patch to the RPU_DATA register.
 - c. Write 0x1 to the RPU_CMND register.
4. Specify the processor behavior when a PC matches a patched ROM address.
 - Trigger a Patch Jump.
 - i. Write 0x1 to the RPU_ADDR register.
 - ii. Write 0 to the bits, in the RPU_DATA register, corresponding to each patch entry that you want to program. For example to trigger a Patch Jump for Patch Entries 0 and 1, write `RPU_DATA[1:0]=00`.
 - iii. Write 0x1 to the RPU_CMND register.
 - Trigger a Patch Breakpoint.
 - i. Write 0x1 to the RPU_ADDR register.
 - ii. Write 1 to the bits, in the RPU_DATA register, corresponding to each patch entry that you want to program. For example,
 - iii. Write 0x1 to the RPU_CMND register.
5. Enable patching of a ROM address.
 - a. Write 0x0 to the RPU_ADDR register.
 - b. Write 1 to the corresponding bit for each patch entry. (For example 0x0000_0001 to enable patching of the first entry) to the RPU_DATA register.
 - c. Write 0x1 to the RPU_CMND register.

37.8 Programming Constraints

- When a delay slot is monitored, ensure that the instructions executed on a patch jump follow all the delay slot illegal instruction limitations.
- A patch table entry should not itself be patched, as this may lead to unpredictable behavior.
- The base address of the ROM must be aligned with the ROM size (defined by `rpu_rom_size`).

37.9 Auxiliary Registers

The RPU provides an indirect programming interface to program the RPU registers. [Table 37-1](#) lists the indirect programming interface registers. [Table 37-3](#) lists the RPU programming registers.

Table 37-1 RPU Indirect Programming Interface Registers

Address	Auxiliary Register Name	Description
0xFC	RPU Build Configuration Register, RPU_BUILD	Build configuration register
0x390	RPU Command Register, RPU_CMND	Initiates reads and writes to the RPU
0x391	RPU Address Register, RPU_ADDR	Address of the internal RPU register to program
0x392	RPU_DATA	Contains the data that you want to read or write to the RPU

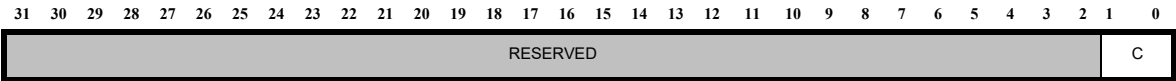
37.9.1 RPU Command Register, RPU_CMND

Address: 0x390

Access: RW

Reset 0x0000_0000

Figure 37-3 RPU_CMND Register



Use this register to perform read or write operations on the RPU programming registers (See [Table 37-3](#)). The read and write operations are performed on the register specified by the RPU_ADDR register and the read/write data is in the RPU_DATA register.

Writing a valid command to the RPU_CMND register triggers that operation in the RPU. Write operations are completed in the RPU before the next instruction is executed by the processor. Reading from the RPU_CMND register returns the last value written to the RPU_CMD register. The RPU_CMND register is cleared on reset.

Table 37-2 RPU_CMND Bit Field Description

Field	Bit	Description
C	[1:0]	Specifies the action performed in the RPU 0x0: RPU_NULL command; perform no action 0x1: RPU_WRITE command, write value from the RPU_DATA register to the address pointed to by the RPU_ADDR register. If the address in RPU_ADDR is not a valid RPU register, the write command is silently ignored. 0x2: RPU_READ command, the data from address pointed by the RPU_ADDR register is stored in the RPU_DATA register. If the address in RPU_ADDR is not a valid RPU register, RPU_DATA is set to 0xffff_ffff. This value is never returned when reading a valid RPU register, and therefore indicates an error. All other values are reserved.

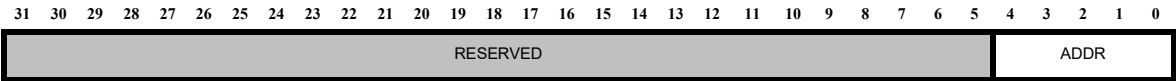
37.9.2 RPU Address Register, RPU_ADDR

Address: 0x391

Access: RW

Reset: 0x0000_0000

Figure 37-4 RPU_ADDR Register



The RPU_ADDR register defines the address of the RPU register that is accessed by an RPU command. See [Table 37-3](#) for a list of valid values for the RPU_ADDR register.

Table 37-3 RPU Programming Registers

RPU_ADDR Value	RPU Programming Register
0x0	RPU_ENABLE
0x1	RPU_MODE
0x2	RPU_STATUS
0x3	RPU_ROM_BASE
0x4	RPU_PATCH_TABLE
RPU Patch Entry registers	
0x10	PatchTable[0].Address
0x11	PatchTable[1].Address
0x12	PatchTable[2].Address
0x13	PatchTable[3].Address
0x14	PatchTable[4].Address
0x15	PatchTable[5].Address
0x16	PatchTable[6].Address
0x17	PatchTable[7].Address
0x18	PatchTable[8].Address
0x19	PatchTable[9].Address
0x1A	PatchTable[10].Address

Table 37-3 RPU Programming Registers

RPU_ADDR Value	RPU Programming Register
0x1B	PatchTable[11].Address
0x1C	PatchTable[12].Address
0x1D	PatchTable[13].Address
0x1E	PatchTable[14].Address
0x1F	PatchTable[15].Address

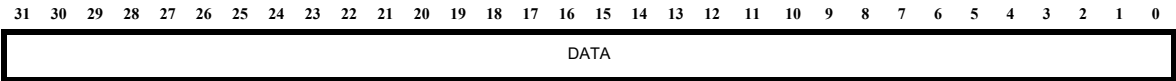
37.9.3 RPU DATA Register, RPU_DATA

Address: 0x392

Access: RW

Reset: 0x0000_0000

Figure 37-5 RPU_DATA Register



This register contains the data read from the RPU or the data to be written to the RPU. The RPU_ADDR register specifies the RPU address that is being accessed.

37.9.4 RPU Enable Register, RPU_ENABLE

RPU_ADDR value	0x0
Reset	0x0000_0000

Figure 37-6 RPU_ENABLE Register



The RPU_ENABLE register implements an individual enable bit for each Patch Entry.

If a Patch Entry has its enable bit set to 1, when the processor attempts to execute an instruction from the corresponding address offset within ROM, either a Patch Jump or Patch Breakpoint is triggered based on the corresponding bit in the RPU_MODE register.

37.9.5 RPU MODE Register, RPU_MODE

RPU_ADDR value	0x1
Reset	0x0000_0000

Figure 37-7 RPU_MODE Register



The RPU_MODE register implements an individual Trap bit for each Patch Entry.

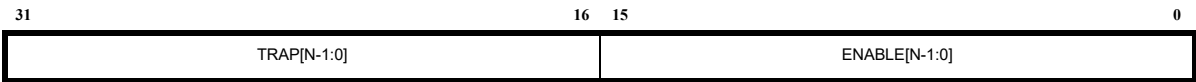
When an enabled Patch Table entry has its Trap bit set to 1 and the processor attempts to execute an instruction from the corresponding address offset within ROM, a Patch Breakpoint is triggered.

When an enabled Patch Table entry has its Trap bit set to 0 and the processor attempts to execute an instruction from the corresponding address offset within ROM, a Patch Jump is triggered.

37.9.6 RPU Status Register, RPU_STATUS

RPU_ADDR value 0x2

Figure 37-8 RPU_STATUS Register



This register indicates the identity of the most recent Patch Entry that has triggered a Patch Breakpoint or a Patch Jump. Each Patch Entry has two bits in this register. For an entry E, bit [E] indicates if the entry has triggered a patch jump. Bit [E+16] indicates if this entry has triggered a breakpoint. Although it is a software error to permit multiple patch entries to share the same address if multiple Patch Events are taken, every taken Patch Event is indicated by a bit being set in the RPU_STATUS register. Note that if any Patch Breakpoint is triggered, no Patch Jumps are triggered.



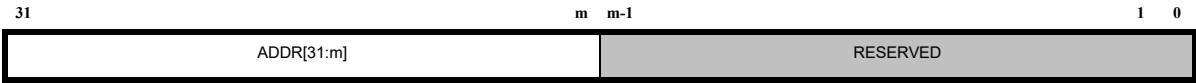
Note

When $N < 16$ (the number of patch entries configured in the processor), The `ENABLE` field can have unused bits from [15: N] and the `TRAP` field can have unused bits from [31: $N+16$].

37.9.7 ROM Base Address Register, RPU_ROM_BASE

RPU_ADDR value	0x3
Reset	0x0000_0000

Figure 37-9 RPU_ROM_BASE Register

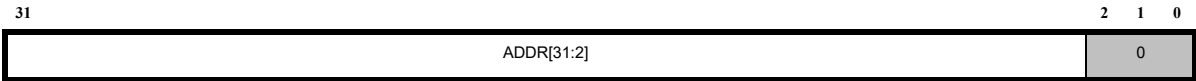


This register holds the size-aligned base address of the ROM. *m* in ADDR[31:*m*] is defined as $\log_2(\text{ROM\_BYTES})$.

37.9.8 RPU Patch Table Base Address Register, RPU_PATCH_TABLE_BASE

RPU_ADDR value	0x4
Reset	0x0000_0000

Figure 37-10 RPU_PATCH_TABLE_BASE Register

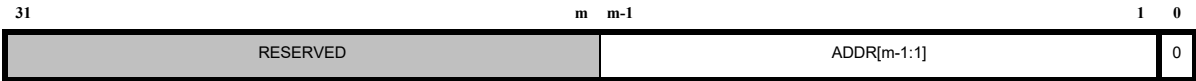


The RPU patch table base address holds the 64-bit aligned address of the memory from where the processor must execute the code when a ROM address is patched. This target address corresponds to the first entry in the Patch Entry. The target address for each Patch Entry exists at 8*patch_number offset from this base address.

37.9.9 RPU Patch Entry Register, PatchTable[n].Address

RPU_ADDR value	0x10 to 0x1F
Reset	0x0000_0000

Figure 37-11 PATCH_OFFSET_n Register



The address must be within the [RPU_ROM_BASE : RPU_ROM_BASE+RPU_ROM_SIZE]. The address field in each Patch Entry implements only those bits required to address the half-word offsets from [0 to (RPU_BUILD.ROM_SIZE - 2)]. The least significant bit is always zero. For example, when ROM_SIZE is 3, the ROM is 128 KB in size and the address field of each Patch Entry uses bits [16:1]. Bit 0 and bits [31:17] are ignored on write (IOW) and read as zero (RAZ). The number of these entry registers available in the RPU is build-time configurable.

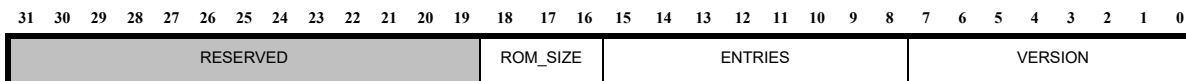
If more than one valid Patch Entry has the same address field, multiple entries can be triggered simultaneously. This is a software error, and RPU behavior is undefined in this case.

37.9.10 RPU Build Configuration Register, RPU_BUILD

Address: 0xFC

Access: R

Figure 37-12 RPU_BUILD Register



This register specifies the RPU version and its build-time configuration.

Table 37-4 RPU_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of RPU 0x01: initial version of the RPU
ENTRIES	[15:8]	Indicates the number of entries (ROM addresses that can be monitored by the RPU) This field contains the value of the <code>-rpu_patch_entries</code> configuration option. When <code>-rpu_patch_entries = 0</code> , it indicates that the RPU is disabled.
ROM_SIZE	[18:16]	Indicates the size of the ROM that can be patched 0x0: 16 KB 0x1: 32 KB 0x2: 64 KB 0x3: 128 KB 0x4: 256 KB 0x5: 512 KB 0x6: 1 MB 0x7: 2 MB This field contains the value of the <code>-rpu_rom_size</code> configuration option.

Part 3

FastMath Extension Pack

In this part:

- [FastMath Data Organization and Addressing](#)
- [FastMath Register Set Extensions](#)
- [FastMath Extension Instruction Details](#)

38

FastMath

The FastMath hardware extensions comprise a small collection of additional instructions, supporting a range of required capabilities, including:

- Basic saturating arithmetic
- Trigonometric functions
- Logarithmic and exponentiation functions
- Fractional division and square root functions

The FastMath component provides base-2 LOG and EXP functions, from which LOG and EXP functions in other bases can be synthesized. If you need to compute Base-10 LOG and EXP operations, you can use the base-2 LOG and EXP functions as explained in the following sections in your processor databook.

- Linear to dB Wrapper Function
- dB to Linear Wrapper Function

38.1 FastMath Data Organization and Addressing

This section describes the data formats used by the ARCV2 ISA.

38.1.1 Fractional Data Formats

The set of extension instructions enabled by the ARCV2 ISA option support 16-bit and 32-bit signed fractional data formats. Signed fractional values are represented in a fixed-point format involving n bits of integer and m bits of fraction. When represented in a k -bit format $n + m = k$, and in the ARCV2 FastMath extension pack k may be 16 or 32. In general, these fixed-point types are denoted nQm , and the most commonly used formats are 1Q15 and 1Q31.

For each nQm format, the minimum interval between two representable values is known as the Unit of the Least Precision (ULP). This is defined as: $ULP = 2^{-m}$. The values of an nQm type are coded using an integer I in the normal range for 2's complement values, that is, -2^{k-1} to $2^{k-1} - 1$. As the binary point is positioned between bits m and $m - 1$, the range of values supported by an nQm type is:

from:

$$NQM\_MIN = \frac{-2^{k-1}}{2^m} = ULP(-2^{k-1})$$

to:

$$NQM\_MAX = \frac{2^{k-1} - 1}{2^m} = ULP(2^{k-1} - 1)$$

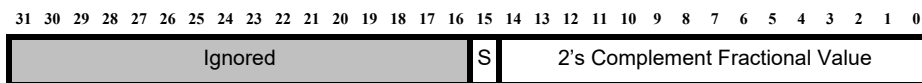
Thus, the mathematical value of the 2's complement integer I , encoded in an nQm format is simply $\frac{I}{2^m}$.

38.1.2 16-Bit Signed Fractions

When a 16-bit signed fraction is contained in a 32-bit register, it is located in the least significant 16 bits. The most significant 16 bits of the 32 bits are ignored by all operations using a 16-bit fraction as a source operand, and when a 16-bit fractional value is written to a destination register, bits [31:16] are cleared.

The FastMath extension pack contains instructions that use 16-bit fractions in 1Q15 format. Of the least significant 16 bits, bit [15] represents the sign, which is effectively also the integer portion, and bits [14:0] represent the fractional portion of the value. The maximum representable value is 0.999969 and the minimum representable value is -1.

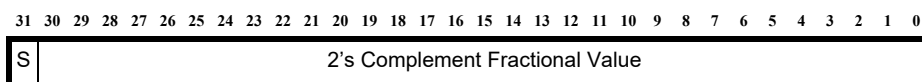
Table 38-1 16-Bit Signed 1Q15 Data Representation in a 32-bit Register



38.1.3 32-Bit Signed Fraction (1Q31)

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.

Table 38-2 32-Bit Signed Fractional Data



38.2 FastMath Register Set Extensions

38.2.1 Extension Auxiliary Registers

Table 38-3 shows a summary of the auxiliary register set defined by the FastMath Pack (FMP) extensions.

The FastMath extension pack contains a single extension auxiliary register.

Table 38-3 DSP Auxiliary Registers

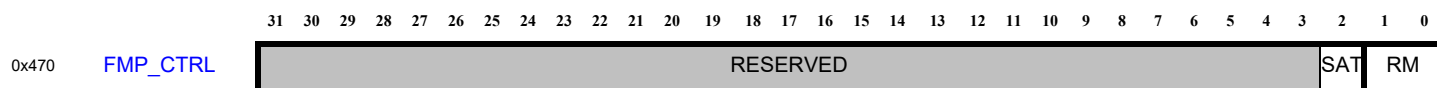
Address	Auxiliary Register Name	Access	Description
0x470	FMP Control Register, FMP_CTRL	rw	Register to control the rounding and guard bit behavior.

38.2.2 FMP Control Register, FMP_CTRL

Address: 0x470

Access: rw

Figure 38-1 FastMath Extension Pack auxiliary Register Map



This register controls the rounding and saturation behavior of FastMath operations, and contains the sticky saturation flag (SAT).

Table 38-4 FMP_CTRL Field Descriptions

Field	Description
RM	Indicates the rounding mode for RNDH instruction 0: no rounding; round down to the nearest representable number 1: truncation; round down to the nearest representable number 2: round up; add 1/2 LSB to the result and truncate 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.
SAT	Sticky saturation status 1: indicates that an FMP operation has saturated its result. This bit is a sticky flag. You must explicitly write a 0 to this bit to clear this flag.

38.2.3 Build Configuration Registers

Table 38-5 Build Configuration Registers

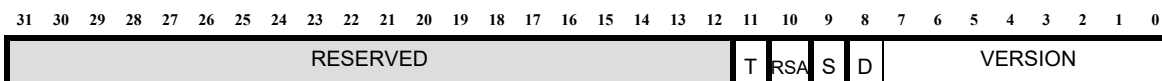
Number	Name	Access	Description
0xE8	FMP Build Configuration Register, FMP_BUILD	R	Build configuration for FMP

38.2.3.1 FMP Build Configuration Register, FMP_BUILD

Address: 0xE8

Access: R

Figure 38-2 FMP_BUILD Register



This configuration register indicates the presence, version and configuration of the FMP component. Applications and software can read this register to detect the presence of FMP, and its configuration. Currently there are no configuration options for the FMP component, and therefore only a Version field is present. There are four individually-selectable instruction group options for the FastMath Pack. The presence of each of these instruction groups is indicated with a single bit (T, R, S and D) within the FMP_BUILD register.

Any combination of T, R S and D bits is permitted except 0000, that is, at least one of the T, R, S and D bits is set to 1, as FMP must be configured with at least one of the four optional instruction groups.

Table 38-6 FMP_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	FMP version number 0x01 = current version
D	8	FMP configuration includes reciprocal and division
S	9	FMP configuration includes square-root
RSA	10	FMP configuration includes RNDH, SATH and ADDS
T	11	FMP configuration includes trig, log and exp functions
Reserved	[31:12]	Reserved. Read as 0 (zero).

38.2.4 Extension Core Registers

The FastMath extension pack does not define any extension core registers.

38.3 FastMath Extension Instruction 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.

All FastMath extension instructions are encoded in 32-bit formats and observe all of the established rules for encoding ARCV2 instructions in 32-bit formats. All extension instructions are implemented in APEX using major opcode 0x07.

All FastMath Pack instructions and register names have the prefix “FMP_” to ensure they do not create any namespace clashes with existing ARCV2 instructions, other customer-defined instructions.

All FastMath Pack instruction encodings and extension register addresses are distinct from existing ARCV2 encodings and addresses.

38.3.1 Semantics of FastMath Arithmetic Operations

The semantics of the extension instructions supported by the FastMath pack are defined in terms of a number of common arithmetic functions and operators. These are defined and referenced in the C++ pseudo-code description of each individual extension instruction later in this chapter.

38.3.2 FastMath fixed-point type declarations

Where the semantic definitions require integer representations larger than 64 bits, the TTMATH library of extended-precision integers is used. TTMATH is a public-domain C++ library, developed under the BSD license and is free for commercial use (<http://www.ttmath.org/>).

As a Q15 type is implemented as a short (2 byte) integer, the normal ABI rules for **short int** apply for argument passing and storage of Q15 types. A short integer is presented in the lower 16 bits of a 32-bit register or literal operand, and is returned in the lower 16 bits of a register operand. This allows such values to be used immediately after being loaded from memory using LDH instructions, and stored back to memory using STH instructions. No alignment shifts are needed in these cases when Q15 values are stored in the least significant 2 bytes of a register.

```

typedef signed short      sint16;
typedef unsigned short    uint16;
typedef signed long       sint32;
typedef unsigned long      uint32;
typedef signed long long   sint64;
typedef unsigned long long uint64;

typedef sint32             q31_t; // declare 1Q31 as 32-bit signed integer
typedef sint16             q15_t; // declare 1Q15 as 16-bit signed integer

// Configuration settings for ttmath
//
#define TTMATH_PLATFORM32
#define TTMATH_NOASM
#define TTMATH_MULTITHREADS
#define TTMATH_POSIX_THREADS
#define TTMATH_NSIM_TYPES
#define TTMATH_NSIM_DISABLE_EXCEPTIONS

#include "ttmath/ttmath.h"

// Declare 128-bit integer types
//
typedef ttmath::Int<4>      sint128;
typedef ttmath::UInt<4>    uint128;

// Conversion from a ttmath type to integer
//
template <typename T, typename S>
inline T ConvertToInt(const S& v) {
    uint64 u;
    v.ToInt(u);
    return static_cast<T>(u);
}

```

The four supported rounding modes are defined thus:

```

typedef enum rnd_mode
{
    RM_NONE      = 0,
    RM_DOWN      = 1,
    RM_UP        = 2,
    RM_UNBIASED  = 3
} rnd_mode_t;

```

38.3.3 Multiplication of extended-precision 37-bit signed integer values

Some of the FMP instructions are implemented using 37x37 bit signed integer multiplications, which deliver a signed 74-bit result. To define bit-accurate semantics, following is an equivalent C++ code for these multiplications.

In most cases, both input values are assumed to be 2Q35 fixed-point fractions, and then the result is a 4Q70 fraction. To accommodate the 37-bit inputs and 74-bit outputs, the arguments are 64-bit signed integers and the output is a 128-bit signed integer.

```
sint128 mult_37x37(sint64 x, sint64 y){ return sint128(x) * y; }
```

38.3.4 Rounding of fractional values

```
sint128 round (    sint128 x,                // value to be rounded
                  sint128 p                // position of the ULP
                  )
{
    sint128 xtra    = sint128(1) << (p-1);
    sint128 mask    = -(sint128_t(1) << p);
    bool     lsb     = (x >> p) & 1;        // bit p from input x
    bool     half_lsb = (x >> (p-1)) & 1;    // bit p-1 from input x
    sint128 rem_lsbs = x & (xtra - 1);        // bits p-2 down to 0 from input x
    sint128 up      = (x + xtra) & mask;      // round-up value = trunc(x + 1/2lsb)
    sint128 down    = x & mask;              // round-down value = trunc(x)

    switch (FMP_CTRL.RM) {
        case RM_UNBIASED:                    // round to nearest even
            if (half_lsb && (lsb || (rem_lsbs != 0)))
                return up;
            else
                return down;
        case RM_UP:                          // round half-up
            if (half_lsb)
                return up;
            else
                return down;
        case RM_DOWN:                        // truncate
            return down;
        case RM_NONE:                       // no rounding
            return x;
    }
}
```

38.3.5 Bit-extraction and rounding functions

The 74-bit fractional result of a 37x37 multiplication is normally reduced back to 2Q35, with optional rounding. Following is the definition that defines how each FMP function rounds and reduces precision.

```

sint64 get_bits_n_downto_m(int n, int m, sint128 value){
    return ConvertToInt<sint64>(sint128(value << (127-n)) >> (127-n+m));
}

sint64 bits_69_33(sint128 value){ return get_bits_n_downto_m(69, 33, value); }
sint64 bits_70_34(sint128 value){ return get_bits_n_downto_m(70, 34, value); }
sint64 bits_70_33(sint128 value){ return get_bits_n_downto_m(70, 33, value); }
sint64 rnd_int38(sint64 value){ return ((value + 1) >> 1); }

sint64 bits_73_37(sint128 value){ return get_bits_n_downto_m(73, 37, value); }
sint64 bits_73_36(sint128 value){ return get_bits_n_downto_m(73, 36, value); }
uint32 bits_32_1(sint128 value) { return ConvertToInt<uint32>(value >> 1); }
sint32 bits_31_0(sint128 value) { return ConvertToInt<sint32>(value); }

sint64 bits_71_35_rnd(sint128 value){
    return rnd_int38(get_bits_n_downto_m(71, 34, value));
}

```


38.3.6 Support functions for trigonometric operations

```

q31_t cos_smallrange (q31_t x, bool invert_result, int iters)
{
    sint64  inp_1Q35, in_sq_1Q35, r37, t37 = 0, result;
    uint32  r32;

    const sint64 coef[7] = {          // cosine coefficients, in 2Q34 format
        sint64(0x00000000100000000),
        sint64(0xFFFFFFFFB10B0CD92),
        sint64(0x00000000103C1F06C),
        sint64(0xFFFFFFFFEAA2C3FC),
        sint64(0x0000000000F0F926),
        sint64(0xFFFFFFFFF966BA),
        sint64(0x00000000000001E26),
    };

    in_1Q35 = (sint64(x) << 32) >> 28; // sign extend and shift binary point

    // First calculate x^2 and clear least-significant bit
    in_sq_1Q35 = bits_69_33(mult_37x37(in_1Q35, in_1Q35));

    // Calculate the polynome
    // iters = 6 for Q31 and 5 for Q15

    for (int i = iters; i > 1; i--) {
        t37 += coef[i];
        t37 = bits_71_35_rnd(mult_37x37(t37, in_sq_1Q35));
    }
    r37 = bits_73_37(mult_37x37((coef[1] + t37), in_sq_1Q35));
    r37 += coef[0];
    r32 = bits_32_1(r37);

    // Invert and saturate result if needed

    if (invert_result == true) {
        result = -r32;
    }
    else {
        if (r32 == 0x80000000)
            result = 0x7fffffff;
        else
            result = r32;
    }
    return result;
}

```

38.3.7 Saturating result values to a defined range

```
bool sat (sint128 x, uint16 p, sint128 *r)
{
    bool neg = (x < 0) ? 1 : 0;
    sint128 mask = -(sint128(1) << p);

    if (neg && ((x & mask) != mask)) {
        *r = mask;
        return true;
    } else if (!neg && ((x & mask) != 0)) {
        *r = ~mask;
        return true;
    }
    *r = x;
    return false;
}
```

38.3.8 Computation of LOG2 (1Q31)

The semantics of the LOG2 function is defined as a numerical approximation based on the Taylor expansion for the log function. The input range for this expansion is [0.5,1.0), which is then compressed by scaling the argument. This is achieved by partitioning the [0.5,1.0) range into 8 equal-sized ranges and multiplying arguments falling each range by a different factor to bring the argument into a narrower range over which the approximation errors are minimized. Following are the factors.

```
#define FLT_2_Q31 (_x_) ((sint32_t)((_x_)*((double)(1<<31))))

static const sint64 fac_a_div2_tb[8] = {
    sint64(0x000000071c71c71c), // FLT_2_Q31((double) 0.8888888888888889 )
    sint64(0x0000000666666666), // FLT_2_Q31((double) 0.8000000000000000 )
    sint64(0x00000005d1745d17), // FLT_2_Q31((double) 0.727272727272727 )
    sint64(0x0000000555555555), // FLT_2_Q31((double) 0.6666666666666667 )
    sint64(0x00000004ec4ec4ec), // FLT_2_Q31((double) 0.615384615384615 )
    sint64(0x0000000492492492), // FLT_2_Q31((double) 0.571428571428571 )
    sint64(0x0000000444444444), // FLT_2_Q31((double) 0.533333333333333 )
    sint64(0x0000000400000000) // FLT_2_Q31((double) 0.500000000000000 )
};
```

After scaling the argument by a factor a and taking logs, the scaling is undone by subtracting $\log_2(a)$. Following are the values of $\log_2(a)$ for each of the 8 regions contained in the table `log2_a_tb[8]`.

```
static const sint64 log2_a_tb[8] = { // rounding term 0x010 already added
    sint64(0xffffffff95c01a3b0), // FLT_2_Q31((double) -0.830074998557688 )
    sint64(0xffffffffa934f098a), // FLT_2_Q31((double) -0.678071905112638 )
    sint64(0xffffffffbacea7c16), // FLT_2_Q31((double) -0.540568381362703 )
    sint64(0xffffffffcae00d1e0), // FLT_2_Q31((double) -0.415037499278844 )
    sint64(0xffffffffd9a8023a2), // FLT_2_Q31((double) -0.299560281858908 )
    sint64(0xffffffffe75767f64), // FLT_2_Q31((double) -0.192645077942396 )
    sint64(0xfffffffff414fdb5a), // FLT_2_Q31((double) -0.093109404391481 )
    sint64(0x00000000000000010) // FLT_2_Q31((double) -0.000000000000000 )
};
```

The Taylor expansion coefficients are nominally defined such that each coefficient i is given by: $\frac{(-1)^{(i+1)}}{2^i \ln 2}$

However, to minimize approximation errors introduced by truncating the sequence at 5 entries, the coefficients are adjusted using a least-squares method. Following is a $\log_2$ approximation function: `fmp_log2_q31`.

```

q31_t log2_ref (q31_t x, int iters)
{
    const sint64 coef[5] = {
        sint64(0x00000005c551e321), // FLT_to_2Q35((double) 0.721347593759739 )
        sint64(0xffffffffd1d5ba202), // FLT_to_2Q35((double) -0.360665067997034 )
        sint64(0x00000001ed25e531), // FLT_to_2Q35((double) 0.240794935777816 )
        sint64(0xffffffffe9b0ae48b), // FLT_to_2Q35((double) -0.174295630014059 )
        sint64(0x0000000187a5331a)  // FLT_to_2Q35((double) 0.191233062030985 )
    };

    const sint64 minus_one = (sint64(-1) << 35); // -1.0 in 2Q35 format

    sint64 r = 0;
    int ind;
    sint64 fac_a_div2, log2_a, sx;
    ind = (x >> 27) & 0x7;

    fac_a_div2 = fac_a_div2_tb[ind];
    sint64 t = sint64(x) << 4;

    sx = bits_71_35_rnd(mult_37x37(t, fac_a_div2));
    sx <= 1;
    sx += minus_one;

    log2_a = log2_a_tb[ind];

    for (int i = iters; i >= 0; i--) {
        r += coef[i];
        r = bits_71_35_rnd(mult_37x37(sx, r));
    }
    r = (r << 1) + log2_a; // rounding term included in log2_a

    return (q31_t)((r >> 4) & 0xffffffff);
}

```

38.3.9 Computation of EXP2 (1Q31)

The semantics of the EXP2 function is defined as a numerical approximation based on the Taylor expansion for the function e^x . Using a change of base, the computation is converted to 2^x . The input range for this function is $[-1.0, 0.0)$, which is then compressed by biasing the argument. This is achieved by partitioning the input range into 8 equal-sized ranges and adding the necessary offset to bring the argument into a narrower range over which the approximation errors are minimized.

The bias values are the multiples of $-1/8$ from 0 to 7, represented exactly in a 1Q31 format, as shown in the table `fac_a_tb[8]`. These are expanded to the appropriate Q-format, by appending zeros, before being added to the input value.

```
static const sint fac_a_tb[8] = {
    0x00000000, // -0/8 in 1Q3 format
    0xf0000000, // -1/8 in 1Q3 format
    0xe0000000, // -2/8 in 1Q3 format
    0xd0000000, // -3/8 in 1Q3 format
    0xc0000000, // -4/8 in 1Q3 format
    0xb0000000, // -5/8 in 1Q3 format
    0xa0000000, // -6/8 in 1Q3 format
    0x90000000 // -7/8 in 1Q3 format
};
```

After computing the exponential function, a correction is made by multiplying the result by 2^y , where y is the previously added offset, starting at 0 and decrementing by 1 for each entry. These power values, for each of the 8 bias offsets, are contained in the table `pow2_a_tb[8]`, scaled to 37-bit fractions for higher precision internal calculations:

```
static const sint64 pow2_a_tb[8] = {
    (sint64(0x7fffffff) << 2), // ((double) 1.0000000000000000) = 2^(-0/8)
    (sint64(0x75606374) << 2), // ((double) 0.917004043204671) = 2^(-1/8)
    (sint64(0x6ba27e65) << 2), // ((double) 0.840896415253715) = 2^(-2/8)
    (sint64(0x62b39509) << 2), // ((double) 0.771105412703970) = 2^(-3/8)
    (sint64(0x5a82799a) << 2), // ((double) 0.707106781186548) = 2^(-4/8)
    (sint64(0x52ff6b55) << 2), // ((double) 0.648419777325505) = 2^(-5/8)
    (sint64(0x4c1bf829) << 2), // ((double) 0.594603557501361) = 2^(-6/8)
    (sint64(0x45cae0f2) << 2) // ((double) 0.545253866332629) = 2^(-7/8)
};
```

The Taylor expansion coefficients are nominally defined such that each coefficient i is given by: $\frac{1}{i! \ln 2}$

However, to minimize approximation errors introduced by truncating the sequence at 5 entries, the coefficients are adjusted using a least-squares method. Following is the `exp2` approximation function `fmp_exp2_q31`. The constant coef table contains the adjusted coefficients, in 2Q35 format.

```

q31_t fmp_exp2_q31(q31_t x)
{
    const sint64 coef[5] = {
        sint64(0x07fffffff0), // FLT_to_2Q35((double) 0.99999999984475385)
        sint64(0x058b90bac0), // FLT_to_2Q35((double) 0.69314714324173410)
        sint64(0x01ebfac6d0), // FLT_to_2Q35((double) 0.24022441217632323)
        sint64(0x0071949da0), // FLT_to_2Q35((double) 0.05545924302356971)
        sint64(0x0012dd4a00), // FLT_to_2Q35((double) 0.00921113779469484)
    };

    if (x == 0)
        return (0x7fffffffL); // 2^0 = 1
    if (x == 0x80000000L)
        return (0x40000000L); // 2^(-1) = 0.5

    int ind = ((-x) >> 28) & 0x7;
    sint64 fac_a = sint64(fac_a_tb[ind]) << 4;
    sint64 t = (sint64(x) << 4) - fac_a;
    sint64 r = coef[4];

    for (int i = 3; i >= 0; i--) {
        r = bits_71_35_rnd(mult_37x37(t, r));
        r += coef[i];
    }
    sint64 pow2_a = pow2_a_tb[ind]; // pow2_a is 4Q33
    sint64 res = bits_69_33(mult_37x37(pow2_a, r));

    return (q31_t)((res + 0x8ULL) >> 4) & 0xffffffffUL);
}

```

38.3.10 Computation of COS (1Q31)

```

q31_t cos_ref(q31_t a, int iters){

    sint32 a_transformed;
    sint32 result;
    bool invert_result;

    // The cosine algorithm uses symmetry of the Cosine-function
    // to improve precision.
    // All possible input-values are mapped on [-.5pi, .5pi),
    // and the coefficients are chosen such that the Taylor expansion
    // is most precise in this interval.
    //
    // The transformation is as follows:
    // [-pi, -.5pi)-> [0, .5pi), inverted
    // [-.5pi, .5pi)-> [-.5pi, .5pi), no inversion
    // [.5pi, pi)-> [-.5pi, 0), inverted

    switch ((a >> 30)&3){
    case 1: // map range [.5pi, pi) on [-.5pi, 0)
        // and invert the result later on
        invert_result = true;
        a_transformed = a + 0x80000000;
        break;
    case 2: // map range [-pi, -.5pi) on [0, .5pi)
        // and invert the result later on
        invert_result = true;
        a_transformed = a - 0x80000000;
        break;
    default: // no transform [-.5pi, .5pi)
        invert_result = false;
        a_transformed = a;
    }

    result = cos_smallrange(a_transformed, invert_result, iters);
    return result;
}

```

38.3.11 Computation of SIN (1Q31)

```
q31_t sin_ref (q31_t a, int iters)
{
    sint32 a_transformed;
    sint32 result;
    bool invert_result = false;

    // The algorithm maps the Sine function on range [-.5pi, .5pi>
    // of the Cosine function, and uses mirroring for the ranges
    // that can not be mapped directly.
    //
    // The mapping is as follows:
    //   [-pi, 0)    -> [-.5pi, .5pi), inverted
    //   [0, pi)    -> [-.5pi, .5pi), no inversion
    //
    if ((a >> 31) & 1){                // [-pi, 0)
        invert_result = true;
        a_transformed = a + 0x40000000;
    }
    else{                               // [0, pi)
        a_transformed = a - 0x40000000;
    }

    result = cos_smallrange(a_transformed, invert_result, iters);
    return result;
}
```


38.3.12 Computation of ATAN (1Q31)

```

q31_t fmp_atan_q31 (q31_t x)
{
    sint64 in_2Q34;
    sint64 in_sq_4Q32;
    sint64 result37;
    sint64 t37 = 0;

    const sint64 coef[11] = {          // atan coefficients, 2Q34 format
        sint64(0x00000000145F306CC),
        sint64(0xFFFFFFFF9359AE62),
        sint64(0x000000000412FF174),
        sint64(0xFFFFFFFFD17846C4),
        sint64(0x00000000023F6DCFA),
        sint64(0xFFFFFFFFE389D938),
        sint64(0x0000000001573A14D),
        sint64(0xFFFFFFFFF219CCC1),
        sint64(0x00000000006D307B0),
        sint64(0xFFFFFFFFFDD7F5B6),
        sint64(0x0000000000051F0F6)
    };

    in_2Q34 = (sint64(x) << 32) >> 29; // sign extend and shift binary point
    in_sq_4Q32 = bits_70_34(mult_37x37(in_2Q34, in_2Q34)); // get x^2
    in_sq_4Q32 = ((in_sq_4Q32 >> 1) << 1); // clear least-sig bit

    for (int i = 10; i > 0; i--) {
        t37 += coef[i];
        t37 = rnd_int38(bits_70_33(mult_37x37(t37, in_sq_4Q32)));
    }
    result37 = rnd_int38(bits_73_36(mult_37x37((coef[0]+t37), in_2Q34)));

    return bits_31_0(result37);
}

```

38.3.13 Computation of SQRT (1Q31)

```
q31_t sqrt_ref (q31_t x, int iters)
{
    sint32 out_root = 0;
    sint64 inp_sample;
    sint64 tmp1;
    sint64 tmp2;
    sint64 tmp_diff;

    inp_sample = sint64(x);
    if (x < 0) {
        inp_sample = -inp_sample;
        if (x == 0x80000000)
            inp_sample = sint64(0x7fffffff);
    }

    // inp_sample and out_root are never negative
    for (int i = 0; i < iters; i++){
        tmp2 = (sint64(out_root) << 2) + 1;
        tmp1 = (inp_sample >> 29) & 0x1fffffff;
        tmp_diff = (tmp1 - tmp2) & 0x1fffffff;
        if (tmp1 >= tmp2){
            inp_sample = (tmp_diff << 29) + (inp_sample & 0x1FFFFFFF);
            inp_sample = inp_sample << 2;
            out_root = (out_root << 1) + 1;
        }
        else{
            inp_sample = inp_sample << 2;
            out_root = (out_root << 1);
        }
    }

    return (out_root << (31-iters));
}
```

38.3.14 1Q31 and 1Q15 Versions of SQRT, LOG2, EXP2, SIN, COS and ATAN

There are 1Q15 and 1Q31 versions of each of the reference functions that compute SQRT, LOG2, EXP2, SIN, COS and ATAN. These are implemented in terms of their 1Q31 equivalent functions, in some cases with iterations of their corresponding approximation algorithm. These functions allow their operators to be performed on native 16-bit data without any requirement to re-align the operand into the most-significant 16 bits of a 32-bit operand. Similarly, the 1Q15 output format allows the results of these functions to be used in any API context where a short (2-byte) value would be expected. Therefore, the results can be stored to memory using a Store Halfword (SH) instruction without explicit re-alignment of data.

The following code shows the specifications for 1Q31 functions in terms of their respective reference functions. These wrapper functions determine the number of iterations required within the approximation algorithms implemented by each reference function.

```
q31_t fmp_sqrt_q31 (q31_t x)
{
    return sqrt_ref(x, 31);
}

q31_t fmp_log2_q31 (q31_t x)
{
    return log2_ref(x, 4);
}

q31_t fmp_cosine_q31 (q31_t x)
{
    return cos_ref(x, 6);
}

q31_t fmp_sine_q31 (q31_t x)
{
    return sin_ref(x, 6);
}
```

Note, the 1Q31 and 1Q15 versions of the ATAN function both require the same number of iterations of the approximation algorithm to attain an accuracy of +/- 1 ULP, and therefore the fmp_atan_q31 function is also the reference function.

All 1Q31 reference functions achieve an accuracy of +/- 1 ULP except for the log2_ref and exp2_ref functions. These achieve accuracies of +/- 2 ULP.

The following code shows the specifications for 1Q15 functions in terms of their respective reference functions. These wrapper functions determine the number of iterations required within the approximation algorithms implemented by each reference function, and the type conversion of the argument and result from 1Q15 to 1Q31, and back to 1Q15.

```
q15_t fmp_sqrt_q15 (q15_t x)
{
    return (sqrt_ref(uint32(x) << 16, 15) >> 16);
}

q15_t fmp_log2_q15 (q15_t x)
{
    return (log2_ref(uint32(x) << 16, 4) >> 16);
}

q15_t fmp_exp2_q15 (q15_t x)
{
    return (fmp_exp2_q31(uint32(x) << 16) >> 16);
}

q15_t fmp_cosine_q15 (q15_t x)
{
    return (cos_ref(uint32(x) << 16, 6) >> 16);
}

q15_t fmp_sine_q15 (q15_t x)
{
    return sin_ref(uint32(x) << 16, 6) >> 16);
}

q15_t fmp_atan_q15 (q15_t x)
{
    return (fmp_atan_q31(uint32(x) << 16) >> 16);
}
```

In each case, a smaller number of approximation iterations are required compared to the 1Q31 versions, while remaining within +/- 1 ULP of result accuracy.

FMP_ADDS

Function

Signed 32-bit addition. The result is saturated.

Extension Group

APEX FastMath Extensions (-support_sat==true)

Operation

$$a = \text{SAT32} (b + c)$$

Instruction Format

op a, b, c

Syntax Example

FMP_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 FMP_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.

Pseudo Code

```
s = sat((sint128) b + c, 31, &a)                                /* FMP_ADDS */
FMP_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

```
FMP_ADDS r0,r1,r2    ; Add contents of r1 with r2 and write result into r0
                     ; and saturate the result if required.
```

Syntax and Encoding

Instruction Code

FMP_ADDS<.f>	a,b,c	00111bbb00100010FBBBCCCCC	AAAAA
FMP_ADDS<.f>	a,b,u6	00111bbb01100010FBBB	uuuuuuAAAAA
FMP_ADDS<.f>	b,b,s12	00111bbb10100010FBBB	ssssssSSSSSS
FMP_ADDS<.cc><.f>	b,b,c	00111bbb11100010FBBBCCCCC	0QQQQQ
FMP_ADDS<.cc><.f>	b,b,u6	00111bbb11100010FBBB	uuuuuu1QQQQQ

FMP_RNDH

Function

Round and saturate a 32-bit signed integer to a 16-bit signed integer

Extension Group

APEX FastMath Extensions (-support_sat==true)

Operation

$b = \text{RND16}(c);$

Instruction Format

op b,c

Syntax Example

FMP_RNDH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of input is set
C		= Unchanged
V	•	= Set if the operation saturates, otherwise cleared

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 FMP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. Status flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
s = sat(round(c,16),31,&t);FMP_CTRL.SAT |= s;          /* FMP_RNDH */
b = t>>16;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = b < 0;
    STATUS32.V = s;
}
```

Assembly Code Example

```
FMP_RNDH r1,r2      ; Round 32-bits in r2 and return saturated value into
                    ; r1.
```

Syntax and Encoding

Instruction Code

FMP_RNDH<.f>	b,c	00111bbb00101111FBBBCCCCC101001
FMP_RNDH<.f>	b,u6	00111bbb01101111FBBBuuuuuu101001

FMP_SATH

Function

Saturate a 32-bit value to a 16-bit value

Extension Group

APEX FastMath Extensions (-support_sat==true)

Operation

$b = \text{SAT16}(c);$

Instruction Format

op b,c

Syntax Example

FMP_SATH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of input is set
C		= Unchanged
V	•	= Set if input is smaller than 0xffff_8000 or bigger than 0x0000_7fff; otherwise cleared

Description

Saturate a 32-bit input fractional value to a 16-bit fractional value. Saturate if bit 15 up to 31 are not equal to the sign bit. The saturation flag FMP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. Status flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
s = sat(c, 15, &b); FMP_CTRL.SAT |= s; /* FMP_SATH */
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = b < 0;
    STATUS32.V = s;
}
```

Assembly Code Example

```
FMP_SATH r1,r2      ; Saturate 32-bits in r2 and return saturated value
                    ; into r1.
```

Syntax and Encoding

Instruction Code

FMP_SATH<.f>	b, c	001111bbb00101111FBBBCCCCC101000
FMP_SATH<.f>	b, u6	001111bbb01101111FBBBuuuuuu101000

FMP_DIVF

Function

Signed 32-bit fractional division.

Extension Group

APEX FastMath Extensions (-support_div==true)

Operation

$$a = (b \ll 31) / c$$

Instruction Format

op a, b, c

Syntax Example

FMP_DIVF<.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

Description

Signed fractional division of the 32-bit operands B and C, assuming $\text{abs}(B) \leq \text{abs}(C)$ or $B == -C$. The fractional quotient gets returned to the destination register. An arithmetic overflow happens if the absolute value of the numerator is larger than the absolute value of the denominator, or if the denominator and numerator are equal, in which case the result is saturated to 0x80000000 or 0x7fffffff and the FMP_CTRL.SAT saturation flag is set to 1.

The overflow flag is set only if the divisor is 0. In case of a division by 0, the result registers are updated with zero values. If the result is non-zero, then the sign of the remainder is always the same as the sign of the B operand. Z, N and V 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

```
if (cc == true) {
    if (c != 0) {
        if ((abs(b) < abs(c)) || (b == -c)) {
            q = (b<<31) / c;
        } else {
            q = (b < 0) == (c < 0) ? 0x7fffffff : 0x80000000;
            FMP_CTRL.SAT = 1;
        }
        a = q;
        if (F == 1) {
            Z_flag = (a == 0) ? 1 : 0;
            N_flag = a < 0;
            V_flag = 0;
        }
    } else {
        a = 0;
        if (F == 1) {
            Z_flag = 1;
            N_flag = 0;
            V_flag = 1;
        }
    }
}
```

Assembly Code Example

```
FMP_DIVF r0,r1,r2      ; Divide Q31 fraction r1 by Q31 fraction r2 and write
                       ; the quotient into r0
```

Syntax and Encoding

Instruction Code		
FMP_DIVF<.f>	a,b,c	00111bbb00100000FBBBCCCCCAAAAAA
FMP_DIVF<.f>	a,b,u6	00111bbb01100000FBBBuuuuuuAAAAAA
FMP_DIVF<.f>	b,b,s12	00111bbb10100000FBBBssssssSSSSSS
FMP_DIVF<.cc><.f>	b,b,c	00111bbb11100000FBBBCCCCC0QQQQQ
FMP_DIVF<.cc><.f>	b,b,u6	00111bbb11100000FBBBuuuuuu1QQQQQ

FMP_DIVF15

Function

Signed 16-bit fractional division.

Extension Group

APEX FastMath Extensions (-support_div==true)

Operation

$$a = (b \ll 15) / c$$

Instruction Format

op a, b, c

Syntax Example

FMP_DIVF15<.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

Description

Signed fractional division of the 16-bit operands B and C, assuming $\text{abs}(B) \leq \text{abs}(C)$ or $B == -C$. The fractional quotient gets returned to the destination register. An arithmetic overflow happens if the absolute value of the numerator is larger than the absolute value of the denominator, or if the denominator and numerator are equal, in which case the result is saturated to 0xffff8000 or 0x00007fff and the FMP_CTRL.SAT saturation flag is set to 1.

The overflow flag is set only if the divisor is 0. In case of a division by 0, the result registers are updated with zero values. If the result is non-zero, then the sign of the remainder is the same as the sign of the B operand. The Z, N and V flags is 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

```

if (cc == true) {
    if (c != 0) {
        if ((abs(b) < abs(c)) || (b == -c)) {
            q = (b<<15) / c;
        } else {
            q = (b < 0) == (c < 0) ? 0x00007fff : 0xffff8000;
            FMP_CTRL.SAT = 1;
        }
        a = q;
        if (F == 1) {
            Z_flag = (a == 0) ? 1 : 0;
            N_flag = a < 0;
            V_flag = 0;
        }
    } else {
        a = 0;
        if (F == 1) {
            Z_flag = 1;
            N_flag = 0;
            V_flag = 1;
        }
    }
}

```

Assembly Code Example

```

FMP_DIVF15 r0,r1,r2    ; Divide Q15 fraction r1 by Q15 fraction r2 and write
                       ; the quotient into r0

```

Syntax and Encoding

Instruction Code

FMP_DIVF15<.f>	a,b,c	00111bbb00100001FBBBCCCCC0AAAAAA
FMP_DIVF15<.f>	a,b,u6	00111bbb01100001FBBBuuuuuu0AAAAAA
FMP_DIVF15<.f>	b,b,s12	00111bbb10100001FBBBssssss0SSSSSS
FMP_DIVF15<.cc><.f>	b,b,c	00111bbb11100001FBBBCCCCC0QQQQQQ
FMP_DIVF15<.cc><.f>	b,b,u6	00111bbb11100001FBBBuuuuuu1QQQQQQ

FMP_RECIP

Function

Signed 32-bit reciprocal.

Extension Group

APEX FastMath Extensions (-support_div==true)

Operation

$$b = (1 \ll 31) / c$$

Instruction Format

op b, c

Syntax Example

FMP_RECIP<.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	•	= Set if divisor is zero

Description

Computes the signed fractional reciprocal of the 32-bit signed integer operand C, when C is not equal to 0 or 1. The fractional reciprocal gets returned to the destination register. An arithmetic overflow happens if C == 1, in which case the result is saturated to 0x7fffffff and the FMP_CTRL.SAT saturation flag is set to 1.

The overflow flag is set only if the divisor C is 0. In case of a division by 0, the result registers are updated with zero values. If the result is non-zero, then the sign of the remainder is always the same as the sign of the B operand. Z, N and V 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

```
if (c != 0) {
    if (c != 1) {
        q = (1<<31) / c;
    } else {
        q = 0x7fffffff;
        FMP_CTRL.SAT = 1;
    }
    a = q;
    if (F == 1) {
        Z_flag = (a == 0) ? 1 : 0;
        N_flag = a < 0;
        V_flag = 0;
    }
} else {
    a = 0;
    if (F == 1) {
        Z_flag = 1;
        N_flag = 0;
        V_flag = 1;
    }
}
```

/* FMP_RECIP */

Assembly Code Example

```
FMP_RECIP r0,r1,r2      ; Compute 1.0/r2 as a Q31 fraction and write
                        ; the quotient into r0 and the remainder into FMP_DATA
```

Syntax and Encoding

Instruction Code		
FMP_RECIP<.f>	b,c	00111 bbb 00 101111 F BBB CCCC CC 101010
FMP_RECIP<.f>	b,u6	00111 bbb 01 101111 F BBB uuuuuu101010

FMP_RECIP15

Function

Signed 16-bit reciprocal.

Extension Group

APEX FastMath Extensions (-support_div==true)

Operation

$$b = (1 \ll 15) / c$$

Instruction Format

op b, c

Syntax Example

FMP_RECIP15<.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	•	= Set if divisor is zero

Description

Computes the signed fractional reciprocal of the 16-bit signed integer operand C, when C is not equal to 0 or 1. The fractional reciprocal gets returned to the destination register. An arithmetic overflow happens if C == 1, in which case the result is saturated to 0x7fff and the FMP_CTRL.SAT saturation flag is set to 1.

The overflow flag is set only if the divisor C is 0. In case of a division by 0, the result registers are updated with zero values. If the result is non-zero, then the sign of the remainder is always the same as the sign of the B operand. Z, N and V 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

```
if (c != 0) {
    if (c != 1) {
        q = (1<<15) /c;
    } else {
        q = 0x7fff;
        FMP_CTRL.SAT = 1;
    }
    a = q;
    if (F == 1) {
        Z_flag = (a == 0) ? 1 : 0;
        N_flag = a < 0;
        V_flag = 0;
    }
} else {
    a = 0;
    if (F == 1) {
        Z_flag = 1;
        N_flag = 0;
        V_flag = 1;
    }
}
```

Assembly Code Example

```
FMP_RECIP15 r0,r1,r2    ; Compute 1.0/r2 as a Q15 fraction and write
                        ; the quotient into r0 and the remainder into FMP_DATA
```

Syntax and Encoding

Instruction Code		
FMP_RECIP15<.f>	b,c	00111bbb00101111FBBBCCCCC101011
FMP_RECIP15<.f>	b,u6	00111bbb01101111FBBBuuuuuu101011

FMP_SQRTF

Function

Compute the square root of a Q31 fractional operand

Extension Group

APEX FastMath Extensions (-support_sqrt==true)

Operation

```
b = sqrt_q31(c);
```

Instruction Format

op b,c

Syntax Example

```
FMP_SQRTF<.f> b,c
```

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Cleared
C		= Unchanged
V	•	= Set if input is less than 0, otherwise cleared

Description

Compute the square root of the Q31 fractional operand C, and return the Q31 result to the destination operand B.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = fmp_sqrt_q31(c);                                /* FMP_SQRTF */
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = 0;
    STATUS32.V = c < 0;
}
```

Assembly Code Example

```
FMP_SQRTF r1,r2      ; Compute square root of Q31 r2 and return Q31 result to r1
```

Syntax and Encoding

Instruction Code

FMP_SQRTF<.f>	b, c	00111 bbb 00101111 FBB CCCCC 100000
FMP_SQRTF<.f>	b, u6	00111 bbb 01101111 FBB uuuuuu 100000

FMP_SQRTF15

Function

Compute the square root of a Q15 fractional operand

Extension Group

APEX FastMath Extensions (-support_sqrt==true)

Operation

`b = sqrt_q15(c);`

Instruction Format

op b,c

Syntax Example

`FMP_SQRTF15<.f> b,c`

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Cleared
C		= Unchanged
V	•	= Set if input is less than 0, otherwise cleared

Description

Compute the square root of the Q15 fractional operand C, and return the Q15 result to the destination operand B.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = fmp_sqrt_q15(c);                                /* FMP_SQRTF15 */
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = 0;
    STATUS32.V = c < 0;
}
```

Assembly Code Example

```
FMP_SQRTF15 r1,r2    ; Compute square root of Q15 r2 and return Q15 result to r1
```

Syntax and Encoding

Instruction Code

FMP_SQRTF15<.f>	b, c	00111bbb00101111FBBBCCCCC100001

FMP_SQRTF15<.f>	b, u6	00111bbb01101111FBBBuuuuuu100001

FMP_COS

Function

Compute the cosine of a Q31 fractional operand

Extension Group

APEX FastMath Extensions (-support_trig==true)

Operation

```
b = cosine_q31(c);
```

Instruction Format

op b,c

Syntax Example

```
FMP_COS<.f> b,c
```

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V		= Unchanged

Description

Compute the cosine function on the Q31 fractional input operand C. The input operand represents an angle expressed in radians divided by Pi. The resulting signed Q31 cosine value is returned to the destination operand B.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = fmp_cosine_q31(c);                                /* FMP_COS */
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b < 0);
}
```

Assembly Code Example

```
FMP_COS r1,r2      ; Compute cosine of Q31 r2 and return Q31 result to r1
```

Syntax and Encoding

Instruction Code

FMP_COS<.f>	b, c	00111 bbb 00 101111 F BBB CCCCC 011110
FMP_COS<.f>	b, u6	00111 bbb 01 101111 F BBB uuuuuu 011110

FMP_COS15

Function

Compute the cosine of a Q15 fractional operand

Extension Group

APEX FastMath Extensions (-support_trig==true)

Operation

```
b = cosine_q15(c);
```

Instruction Format

op b,c

Syntax Example

```
FMP_COS15<.f> b,c
```

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V		= Unchanged

Description

Compute the cosine function on the Q15 fractional input operand C. The input operand represents an angle expressed in radians divided by Pi. The resulting signed Q15 cosine value is returned to the destination operand B.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = fmp_cosine_q15(c);                                /* FMP_COS15 */
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b < 0);
}
```

Assembly Code Example

```
FMP_COS15 r1,r2      ; Compute cosine of Q15 r2 and return Q15 result to r1
```

Syntax and Encoding

Instruction Code

FMP_COS15<.f>	b, c	00111 bbb 00 101111 F BBB CCCCC 101100
FMP_COS15<.f>	b, u6	00111 bbb 01 101111 F BBB uuuuuu 101100

Assembly Code Example

```
FMP_SIN r1,r2      ; Compute sine of Q31 r2 and return Q31 result to r1
```

Syntax and Encoding

Instruction Code

FMP_SIN<.f>	b, c	001111bbb00101111FBBBCCCCC011111
FMP_SIN<.f>	b, u6	001111bbb01101111FBBBuuuuuu011111

FMP_SIN15

Function

Compute the sine of a Q15 fractional operand

Extension Group

APEX FastMath Extensions (-support_trig==true)

Operation

```
b = sine_q15(c);
```

Instruction Format

op b,c

Syntax Example

```
FMP_SIN15<.f> b,c
```

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V		= Unchanged

Description

Compute the sine function on the Q15 fractional input operand C. The input operand represents an angle expressed in radians divided by Pi. The resulting signed Q15 sine value is returned to the destination operand B.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = fmp_sine_q15(c);                                /* FMP_SIN15 */
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b < 0);
}
```

Assembly Code Example

```
FMP_SIN15 r1,r2      ; Compute sine of Q15 r2 and return Q15 result to r1
```

Syntax and Encoding

Instruction Code

FMP_SIN15<.f>	b, c	00111 bbb 00 101111 F BBB CCCCC 101101
FMP_SIN15<.f>	b, u6	00111 bbb 01 101111 F BBB uuuuuu 101101

FMP_ATAN

Function

Compute the arctangent of a Q31 fractional operand

Extension Group

APEX FastMath Extensions (-support_trig==true)

Operation

```
b = atan_q31(c);
```

Instruction Format

op b,c

Syntax Example

```
FMP_ATAN<.f> b,c
```

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V		= Unchanged

Description

Computes the inverse tangent function on the Q31 fractional input operand C. 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.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = fmp_atan_q31(c);                                     /* FMP_ATAN */
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b < 0);
}
```

Assembly Code Example

```
FMP_ATAN r1,r2      ; Compute arctangent of Q31 r2 and return Q31 result to r1
```

Syntax and Encoding

Instruction Code

FMP_ATAN<.f>	b, c	00111bbb00101111FBBBCCCCC100101
FMP_ATAN<.f>	b, u6	00111bbb01101111FBBBuuuuuu100101

FMP_ATAN15

Function

Compute the arctangent of a Q15 fractional operand

Extension Group

APEX FastMath Extensions (-support_trig==true)

Operation

```
b = atan_q15(c);
```

Instruction Format

op b,c

Syntax Example

```
FMP_ATAN15<.f> b,c
```

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V		= Unchanged

Description

Computes the inverse tangent function on the Q15 fractional input operand C. 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.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = fmp_atan_q15(c);                                     /* FMP_ATAN15 */
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b < 0);
}
```

Assembly Code Example

```
FMP_ATAN15 r1,r2      ; Compute arctangent of Q15 r2 and return Q15 result to r1
```

Syntax and Encoding

Instruction Code

```
FMP_ATA15N<.f>    b, c    00111bbb00101111FBBBCCCCC101110
```

```
FMP_ATAN15<.f>    b, u6    00111bbb01101111FBBBuuuuuu101110
```

FMP_EXP2

Function

Compute the value 2^x for a given Q31 fractional operand x , when x is in the range $[-1,0)$.

Extension Group

APEX FastMath Extensions (`-support_trig==true`)

Operation

`b = exp2_q31(c);`

Instruction Format

`op b,c`

Syntax Example

`FMP_EXP2<.f> b,c`

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if the fractional Q31 operand is not in the range $[-1, 0)$

Description

Computes the base-2 exponentiation of the Q31 fractional input operand C , when C is in the range $[-1,0)$. If the operand C is not in the range $[-1,0)$, then a zero result is returned. The result $B = 2^C$ is a Q31 fraction in the range $[0.5,1)$. If the `.F` bit is set, the `Z` flag and the `V` flag are set when the input is out of range, and the `N` flag is set when the result is negative.

An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

in_range = q31_t(c) <= 0x00000000;                                /* FMP_EXP2 */
if (in_range)
    b = fmp_exp2_q31(c);
else
    b = 0;
if (F) {
    STATUS32.Z = b == 0x0000_0000; // equiv. to !in_range
    STATUS32.N = (b < 0);
    STATUS32.V = !in_range;
}

```

Assembly Code Example

```

FMP_EXP2 r1,r2           ; Compute 2 to the power of the Q31 r2 argument
                        ; and return the Q31 result to r1

```

Syntax and Encoding

Instruction Code

FMP_EXP2<.f>	b, c	00111 bbb 00 101111 F BBB CCCCC 100111
FMP_EXP2<.f>	b, u6	00111 bbb 01 101111 F BBB uuuuuu 100111

FMP_EXP215

Function

Compute the value 2^x for a given Q15 fractional operand x , when x is in the range $[-1,0)$.

Extension Group

APEX FastMath Extensions (`-support_trig==true`)

Operation

`b = exp2_q15(c);`

Instruction Format

`op b,c`

Syntax Example

`FMP_EXP215<.f> b,c`

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if the fractional Q31 operand is not in the range $[-1, 0)$

Description

Computes the base-2 exponentiation of the Q15 fractional input operand C , if C is in the range $[-1,0)$. If the operand C is not in the range $[-1,0)$, then a zero result is returned. The result $B = 2^C$ is a Q15 fraction in the range $[0.5,1)$. If the `.F` bit is set, the `Z` flag and the `V` flag are set when the input is out of range, and the `N` flag is set when the result is negative.

An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

/* FMP_EXP215 */
in_range = q15_t(c) <= 0;
if (in_range)
    b = fmp_exp2_q15(c);
else
    b = 0;
if (F) {
    STATUS32.Z = b == 0x0000_0000; // equiv. to !in_range
    STATUS32.N = (b < 0);
    STATUS32.V = !in_range;
}

```

Assembly Code Example

```

FMP_EXP215 r1,r2      ; Compute 2 to the power of the Q15 r2 argument
                      ; and return the Q15 result to r1

```

Syntax and Encoding

Instruction Code

FMP_EXP215<.f>	b, c	00111 bbb 00 101111 F BBB CCCCC 101111
FMP_EXP215<.f>	b, u6	00111 bbb 01 101111 F BBB uuuuuu 101111

FMP_LOG2

Function

Compute the base-2 logarithm of the Q31 fractional operand x , when x is in the range $[0.5, 1)$.

Extension Group

APEX FastMath Extensions (`-support_trig==true`)

Operation

$b = \log2\_q31(c);$

Instruction Format

op b,c

Syntax Example

FMP_LOG2<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if the fractional Q31 operand is not in the range $[0.5, 1)$

Description

Computes the base-2 logarithm of the Q31 fractional input operand C , when C is in the range $[0.5, 1)$. If the operand C is not in the range $[0.5, 1)$, then a zero result is returned. The result is a Q31 fraction in the range $[-1, 0)$. If the .F bit is set, the Z flag and the V flag are set when the input is out of range, and the N flag is set when the result is negative.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

in_range = q31_t(c) >= 0x40000000;           /* FMP_LOG2 */
if (in_range)
    b = fmp_log2_q31(c);
else
    b = 0;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b < 0);
    STATUS32.V = !in_range;
}

```

Assembly Code Example

```

FMP_LOG2 r1,r2      ; Compute log2 of the Q31 r2 argument and return the
                    ; Q31 result to r1

```

Syntax and Encoding

Instruction Code

FMP_LOG2<.f>	b, c	00111 bbb 00 101111 F BBB CCCCC 100110
FMP_LOG2<.f>	b, u6	00111 bbb 01 101111 F BBB uuuuuu 100110

FMP_LOG215

Function

Compute the base-2 logarithm of the Q15 fractional operand x , when x is in the range $[0.5,1)$.

Extension Group

APEX FastMath Extensions (`-support_trig==true`)

Operation

$b = \log2\_q15(c);$

Instruction Format

op b,c

Syntax Example

FMP_LOG2<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if the fractional Q31 operand is not in the range $[0.5, 1)$

Description

Computes the base-2 logarithm of the Q15 fractional input operand C , when C is in the range $[0.5,1)$. If the operand C is not in the range $[0.5,1)$, then a zero result is returned. The result is a Q15 fraction in the range $[-1,0)$. If the .F bit is set, the Z flag and the V flag are set when the input is out of range, and the N flag is set when the result is negative.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

in_range = (q15_t(c) >= 0x4000);           /* FMP_LOG215 */
if (in_range)
    b = fmp_log2_q15(c);
else
    b = 0;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b < 0);
    STATUS32.V = !in_range;
}

```

Assembly Code Example

```

FMP_LOG215 r1,r2    ; Compute log2 of the Q15 r2 argument and return the
                    ; Q15 result to r1

```

Syntax and Encoding

Instruction Code

FMP_LOG215<.f>	b, c	00111 bbb 00 101111 F BBB CCCCC 110000
FMP_LOG215<.f>	b, u6	00111 bbb 01 101111 F BBB uuuuuu 110000

Part 4

ARCV2 Scalar DSP Extensions

In this part:

- [Scalar DSP Introduction](#)
- [Data Organization and Addressing](#)
- [DSP Auxiliary Interface](#)
- [Scalar DSP Instruction Set Details](#)

38

Scalar DSP Introduction

This document is intended for programmers of the ARCV2 ISA (Instruction-Set Architecture) and the ARCV2DSP ISA. The ARCV2 ISA comprises a mandatory set of baseline features, together with a collection of optional extensions to the ISA. This document covers all aspects of the ARCV2 ISA.

The ARCV2DSP is an extension of the ARCV2 ISA. The ARCV2DSP includes over 100 new DSP instructions for accelerating signal processing algorithms, including vector and complex MUL/MAC operations. The ARCV2DSP includes:

- a unified, single-cycle 32×32 MUL/MAC unit with 40-bit/72-bit accumulators.
- fractional support (Q31 and Q15 data types) with saturating arithmetic, rounding and non-rounding instructions, as well as divide and square root instructions to support filtering, fast Fourier transform (FFT), and other signal processing algorithms.
- vector operations to process multiple data values in a single operation.

38.1 Key Features of the ARCV2 ISA DSP Extensions

- Basic Saturating Arithmetic instructions
- Vector Unpacking instructions
- Vector ALU instructions
- Accumulator instructions
- Vector SIMD 16×16 MAC instructions
- Dual Inner Product 16×16 MAC instructions
- 32×32 and 16×16 MAC instructions
- Dual 16×8 MAC instructions
- Single-cycle Complex multiply, conjugated multiply, and FFT instructions
- 32×16 MAC instructions
- Accumulator shifting allowed in the $[-72:72]$ range.
- A small address generation unit (AGU)

38.2 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.

38.3 Configurability

The ARCV2 architecture offers several well-defined methods for configuring the programming model and instruction set. In addition, each processor within the ARCV2 family offers well-defined methods for configuring additional core component, and to select from the available micro-architectural implementation options to choose an appropriate trade-off between performance, complexity, operating frequency, and energy consumption.

Table 38-1 DSP Configuration Options

Option Name	ARCHitect Option	Option Values	Description
DSP			
Implementation	-dsp_impl	inferred optimized	Infer the datapath components from Verilog or optimized using carry-save components for timing.
ITU Support	-dsp_itu	false true	Include support for pre-accumulate shifting required by ITU codecs ■ true: supported ■ false: insupported
Divide and Square Root	-dsp_divsqrt	none radix4	Enable support for divide and square-root instructions. radix4 computes two result bits per clock cycle
Wide DSP support	-dsp_wide	false true	If true, enable DSP instructions with 64 source operands.

39

Data Organization and Addressing

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

39.1 DSP Data Formats

The set of DSP extension instructions, enabled by the HAS_DSP option, use the following data formats.

39.1.1 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 39-1 8-Bit Signed Integer Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	2's complement							s	2's complement							s	2's complement							s	2's complement						

39.1.2 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 39-2 16-Bit Signed Integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ignored																s	2's complement														

39.1.3 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 39-3 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															

39.1.4 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 39.1.7. Operations involving scalar 16-bit fractional source operands come in two versions:

- Instructions whose mnemonic has suffix 'L', take a Q15 scalar operand from the least significant 16 bits of a 32-bit operand, for example, MPYWHFL.
- Instructions whose mnemonic has suffix 'M', take a Q15 scalar operand from the most significant 16 bits of a 32-bit operand, for example, MSUBWHFM.

The maximum representable value for a 16-bit signed fraction is 0.999969 and minimum value is -1.0.

39.1.5 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 39-4 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															

39.1.6 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 39-5 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															

39.1.7 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 39-6 16-Bit Signed Fractional Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	2's Complement Fractional Value															S	2's Complement Fractional Value														

39.1.8 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 39-7 Complex 16+16 Bit Complex Signed Fractional Data

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	2's Complement Imaginary Fractional Value															S	2's Complement Real Fractional Value														

39.1.9 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 39-8 32-Bit Signed Integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	2's complement																														

39.1.10 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 39-9 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																															

39.1.11 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 39-10 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																													

39.1.12 Accumulators

The DSP component uses the following auxiliary registers for DSP operations:

- “ACC0_LO” on page 1432
- “ACC0_HI” on page 1434
- “ACC0_GLO” on page 1433
- “ACC0_GHI” on page 1435

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.

39.1.12.1 Vector Accumulators

The ACC0_LO, ACC0_HI, ACC0_GLO and ACC0_LO, and ACC0_GHI and ACC0_HI are the vector accumulators. The format of these accumulators is dependent on the following configuration options:

- Whether the guard bits are enabled (DSP_CTRL==1)
- Whether pre-accumulation is enabled (DSP_CTRL.PA==1)

The vector accumulators are used in the following operations:

Operation	ACC0_GHI GUARD	ACC0_HI	ACC0_GLO GUARD	ACC0_LO
2x32 bits		val1[31:0]		val0[31:0]
1x64 bits		val[63:32]		val[31:0]
1x32b		val[31:0]		
1x40b	val[39:32]	val[31:0]		
2x40b	val1[39:32]	val1[31:0]	val0[39:32]	val0[31:0]

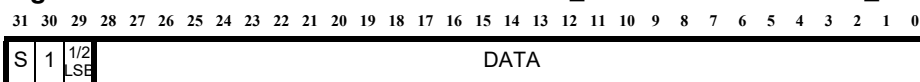


Note

The 1x72 bit operation is explained in section “Wide Accumulator” on page 1427.

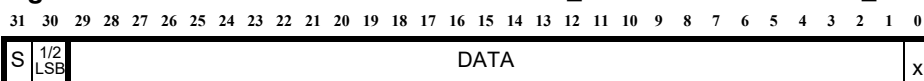
When DSP_CTRL.GE==0 and DSP_CTRL.PA==0, [Figure 39-1](#) represents the vector accumulators (ACC0 LO and ACC0 HI).

Figure 39-1 Vector Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==0



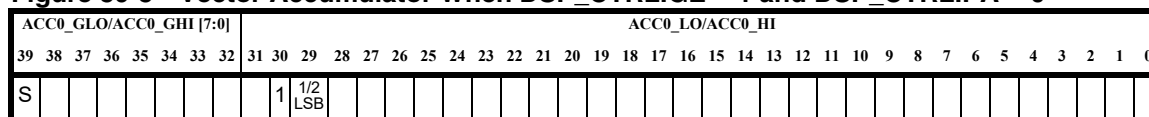
When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`, [Figure 39-2](#) represents the vector accumulators (`ACC0_LO` and `ACC0_HI`). In this mode, the LSB is set only on saturation.

Figure 39-2 Vector Accumulator When DSP_CTRL.GE==0 and DSP_CTRL.PA==1



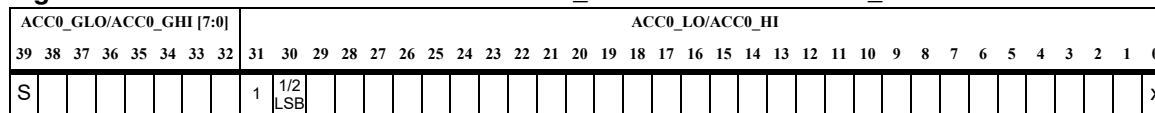
When DSP_CTRL.GE==1 and DSP_CTRL.PA==0, [Figure 39-3](#) represents the vector accumulators.

Figure 39-3 Vector Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==0



When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`, [Figure 39-4](#) represents the vector accumulators. In this mode, the LSB is set only on saturation.

Figure 39-4 Vector Accumulator When DSP_CTRL.GE==1 and DSP_CTRL.PA==1



39.1.12.2 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:

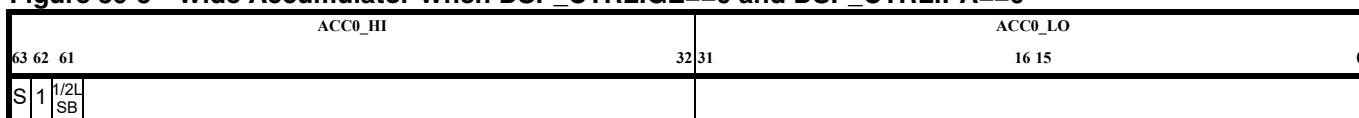
Operation	ACC0_GHI GUARD	ACC0_HI	ACC0_LO
1x72 bits	val[71:64]	val[63:32]	val[31:0]

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 39-5](#) represents the wide accumulator.

Figure 39-5 Wide Accumulator When DSP CTRL.GE==0 and DSP CTRL.PA==0



When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`, [Figure 39-6](#) represents the wide accumulator. In this mode, the LSB is set only on saturation.

Figure 39-6 Wide Accumulator When `DSP_CTRL.GE==0` and `DSP_CTRL.PA==1`

ACC0_HI															
63	62	61													0
S	1/2	LSB													X

When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==0`, [Figure 39-7](#) represents the wide accumulator.

Figure 39-7 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										1/2										LSB									

When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`, [Figure 39-8](#) represents the wide accumulator.

Figure 39-8 Wide Accumulator When `DSP_CTRL.GE==1` and `DSP_CTRL.PA==1`

ACC0_GHI [7:0]										ACC0_HI										ACC0_LO																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																						
71										63 62 61										3231										16 15										0																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																		
S										1	1/2	LSB																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														

39.1.13 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.

39.1.14 Accumulator Operations

[Table 39-11](#) provides the list of instructions that you can use to shift, set, read, divide, and compute the square root of the accumulators:

Table 39-11 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 39-11 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

40

DSP Auxiliary Interface

40.1 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 40-1 DSP Auxiliary Registers

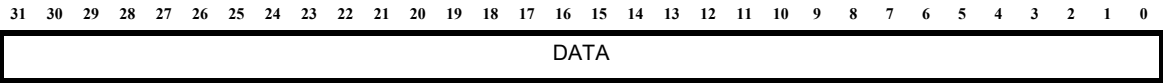
Address	Auxiliary Register Name	Description
0x580	Accumulator Low Register, ACC0_LO	Low accumulator register.
0x581	Low Accumulator Guard and Status Register, ACC0_GLO	Low accumulator guard and status register.
0x582	Accumulator High Register, ACC0_HI	High accumulator register.
0x583	High Accumulator 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.

40.1.1 Accumulator Low Register, ACC0_LO

Address: 0x580

Access: rw

Figure 40-1 ACC0_LO Register



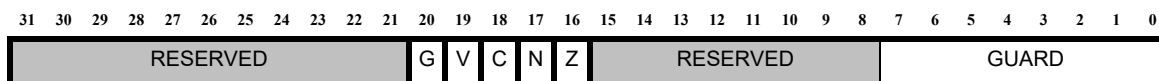
This register holds the lower 32 bits of the result from a vector accumulate operation. For complex math operations, this register holds the real part of a complex operand in a little-endian mode and the imaginary part of the complex operand in a big-endian mode. For operations that involve a wide accumulator, this register holds the least significant 32 bits of the wide accumulator.

40.1.2 Low Accumulator Guard and Status Register, ACC0_GLO

Address: 0x581

Access: rw

Figure 40-2 ACC0_GLO Register



This register holds the guard bits and the status flags of the ACC0_LO accumulator. This register is not used in the wide accumulator operations.

Table 40-2 ACC0_GLO Field Descriptions

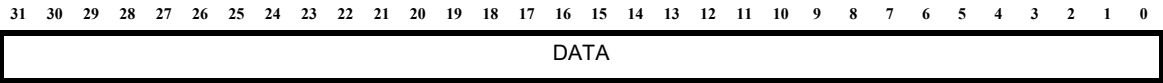
Field	Bit	Description
GUARD	[7:0]	Guard bits; overflow bits of the low vector accumulator. For example, if a the low vector accumulator (ACC0_LO) overflows, this field stores the overflow bits (up to 8 bits). This field is valid only if DSP_CTRL.GE==1.
Z	[16]	Zero flag; this bit is set only by the fractional 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).

40.1.3 Accumulator High Register, ACC0_HI

Address: 0x582

Access: rw

Figure 40-3 ACC0_HI Register



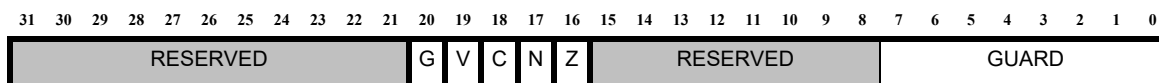
This register holds the higher 32 bits of the result from a vector accumulate operation. For complex math operations, this register holds the imaginary part of a complex operand in a little-endian mode and the real part of the complex operand in a big-endian mode. For operations that involve a wide accumulator, this register holds the most significant 32 bits of the wide accumulator.

40.1.4 High Accumulator Guard and Status Register, ACC0_GHI

Address: 0x583

Access: rw

Figure 40-4 ACC0_GHI Register



This register holds the guard bits and the status flags for the ACC0_HI accumulator.

Table 40-3 ACC0_GHI Field Descriptions

Field	Bit	Description
GUARD	[7:0]	Guard bits; overflow bits of the high vector accumulator. For example, if a the low vector accumulator (ACC0_HI) overflows, this field stores the overflow bits (up to 8 bits). This field is valid only if DSP_CTRL.GE==1.
Z	[16]	Zero flag; this bit is set only by the fractional MAC instructions when the accumulator (ACC0_HI) is zero.
N	[17]	Negative status flag; set only by the fractional MAC instructions when the accumulator (ACC0_HI) is negative.
C	[18]	Carry status flag; this carry flag is always cleared (0).
V	[19]	Overflow status flag; set only by the fractional MAC instructions if the accumulator (ACC0_HI) has overflown on the most recent instruction.
G	[20]	Indicates if the guard bits are actively used in the fractional instructions. A fractional MAC operation sets this bit only if the GUARD field bits are not equal to the sign bit of the accumulator (ACC0_HI).

40.1.5 DSP Butterfly Instructions Data Register, DSP_BFLY0

Address: 0x598

Access: rw

Figure 40-5 DSP_BFLY0 Register in Little-Endian Mode

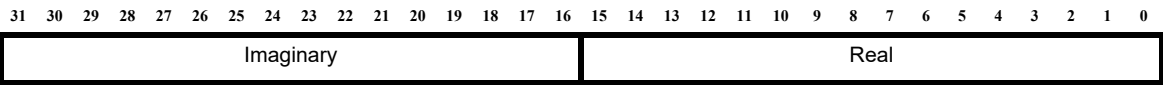
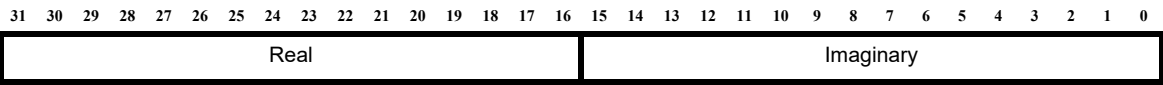


Figure 40-6 DSP_BFLY0 Register in Big-Endian Mode



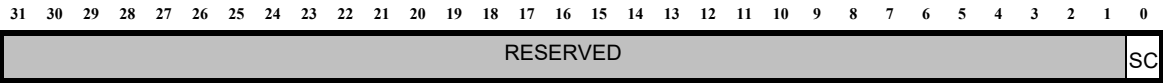
This register stores the implicit complex A0 operand that is used in the butterfly instructions. The DSP_BFLY0 register consists of the real and imaginary components of a 16+16 bit complex fractional number. The CBFLYH1R instruction can modify this register. The CBFLYH0R and CBFLYH1R instructions can implicitly read this register. This register is present in the processor build only if has_dsp==true.

40.1.6 DSP FFT Control Register, DSP_FFT_CTRL

Address: 0x59E

Access: rw

Figure 40-7 DSP_FFT_CTRL Register



This register controls the scaling behavior of the complex FFT butterfly instructions.

This register is present in the processor build only if `has_dsp==true`.

Table 40-4 DSP_FFT_CTRL Field Descriptions

Field	Bit	Description
SC	[0]	Controls the scaling of the complex results 0x0: no scaling 0x1: the butterfly instructions (CBFLYHF0R and CBFLYH1R) scale the complex result by right-shifting the result by one bit.

40.1.7 DSP Control Register, DSP_CTRL

Address: 0x59F

Access: rw

Figure 40-8 DSP_CTRL Register



This register controls the rounding and guard bit behavior of the DSP multiply and accumulate operations.

Table 40-5 DSP_CTRL Field Descriptions

Field	Bit	Description
RM	[1:0]	Indicates the rounding mode 0x0/1: round down to the nearest representable number 0x2: round up; add 1/2 LSB to the result and truncate 0x3: convergent rounding; rounding to the nearest even. Convergent rounding is the same as rounding up except in the following case: When the 1/2 LSB is set and all lower significant bits are cleared, convergent rounding rounds up only if the LSB is set.
GE	[2]	Enable guard bits 0x0: disable guard bits; the wide accumulator saturates at 64 bits and the vector accumulators saturate at 32 bits 0x1: enable guard bits; the wide accumulator saturates at 72 bits and the vector accumulators saturate at 40 bits
PA	[3]	Enable pre-accumulation shift mode If <code>dsp_itu==true</code> 0x0: post accumulation. The fractional product is not shifted before accumulation. The accumulator is of type 1Q30. 0x1: pre accumulation. The fractional product is shifted by one position to the left before accumulation. The accumulator value in this case is of type Q31 with <code>LSB==0</code>. If <code>dsp_itu==false</code> - The PA bit is fixed to 0; pre-accumulation shift mode is not supported.

Table 40-5 DSP_CTRL Field Descriptions (Continued)

Field	Bit	Description
SE	[4]	Enable saturation for fractional instructions 0x0: disable saturation for guarded operations with fractional instructions 0x1: enable saturation for guarded operations with fractional instructions
SAT	[16]	Sticky saturation status 1: indicates that a DSP operation is saturated in one of the following cases: - Accumulator saturation - Result saturation - Multiply pre-accumulate saturation when <code>DSP_CTRL.GE==0</code> and <code>DSP_CTRL.PA==1</code> This bit is a sticky flag. You must explicitly write a 0 to this bit to clear this flag.

40.2 Build Configuration Registers

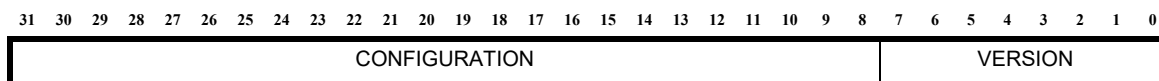
The embedded software or host debug software can use a reserved set of auxiliary registers, called Build Configuration Registers (BCRs), to detect the configuration of the ARCV2-based system. The build configuration registers identify the version of each extension and also specific configuration information.

40.2.1 DSP Build Configuration Register, DSP_BUILD

Address: 0x7A

Access: R

Figure 40-9 DSP_BUILD Register



This configuration register indicates the version and configuration of the DSP component. Applications and software can read this register to detect the DSP configuration.

Table 40-6 DSP_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	DSP version number 0x22: current version
CONFIGURATION	[9:8]	Indicates support for the divide and square-root instructions 0x0: DIV/REM instructions are not supported. 0x2: Radix-4 fast DIV, DIVU, REM and REMU implementation.
CONFIGURATION	[10]	Indicates support for the single-cycle complex instructions This field is fixed to 0x1.
CONFIGURATION	[12:11]	Indicates support for the accumulator shift This bit is fixed 0x2; that is shift accumulator in the range [-72:72].
CONFIGURATION	[13]	Indicates support for the ITU pre-accumulate support. 0x0: Preaccumulate shifting hardware is not supported. 0x1: Preaccumulate shifting hardware is supported
CONFIGURATION	[14]	Indicates support for the wide instructions. 0x0: Wide instructions are not supported. 0x1: Wide instructions are supported.
CONFIGURATION	[31:15]	Reserved. Read as 0.

41

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.

41.1 Address Generation

To generate an address, an AGU returns the pointer value and updates the associated address pointer. The address pointer can be used as the address operand in load/store operations. The width of the LD/ST access is implied by the optional suffix (B/H/D) of the LD/ST operation.

If an unaligned AGU-based LD/ST of data straddles the wrap boundary, the lower part of the data maps to the top of the modulo buffer and the upper part of the data maps to the bottom of the modulo buffer. For example, if a modifier specifies a modulo buffer of 32 bytes, starting at address 0x800, an unaligned 32 byte LD from address 0x81e using that modifier returns [0x801][0x800][0x81f][0x81e].

Unaligned AGU-based LD/ST instructions straddling the wrap boundary in ICCM result in undefined behavior.

[Example 41-1](#) defines the address updates.

Example 41-1 Pseudocode for Updating an Address Pointer Value

```
// modifier fields
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 = number 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;
    }
    if (offset == 0) {
        // no update, return
        return;
    }

    // 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 lsb
        lsbmask = -power(2, mod.vw);
        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;
}

```

[Example 41-2](#) shows the ARC HS assembly code using address generators.

Example 41-2 Assembly Code Using Address Generators

```

LD r1, [agu_u1]      ; load from address pointer and update
LD r2, [agu_u2]      ; load from address pointer and update
VPERM r1, r1, s12    ; optional data format conversion (byte permute and sign extend)
VPERM r2, r2, s12    ; optional data format conversion (byte permute and sign extend)
MPY r0, r1, r2        ; regular multiply
VPERM r0, r0, s12    ; optional data format conversion (byte permute and sign extend)
ST r0, [agu_u0]      ; store to address pointer and update

```

The ARC HS code uses the AGU to generate the pointer only; data transformations are handled by explicit DSP packing/unpacking instructions and memory accesses are handled by explicit load/store operations.

41.2 Limitations

- AGU-extended core registers can only be used in the following instructions MODAPP, LD, LDB, LDH, LDD, ST, STB, STH, STD. Using them with other instructions raises an illegal-operand exception.
- The ARCV2 architecture supports a limited number of extended core registers, which are shared between different optional units, so allocating AGU registers restricts other system features, such as extended core registers available for APEX. If AGU is enabled, APEX extensions cannot define extended core registers used by AGU. If AGU size is small, AGU uses core registers r32 to r35 and APEX can define registers r36 up to r57 as r58 to r59 are used by DSP accumulators.
- For load/store instructions, AGU-extended core registers can only be used as a load register with offset format (F32_LD_OFFSET) or as a store register with offset format (F32_ST_OFFSET).
- AGUs can only be used as the B operand in the load/store instructions; using AGUs as an A or C operand raises an illegal-operand exception.
- The load/store offset (*s9*) must be equal to 0; post-increment is implied by modifier; a non-zero *s9* offset raises an illegal operand exception.
- The load/store *aa* fields need to be equal to 0, that is, no increment mode; other *aa* values raise an illegal-operand exception.
- Only the second operand C in the MODAPP instruction can be an AGU register; the destination operand needs to be 0 (void); using non-0 as the B operand raises an illegal-operand exception.
- The modifier address-register pointers and optional wrap-register pointer must point to an existing register; pointers to non-existing registers raise an illegal operand exception.

- An AGU access straddling a modulo boundary that is not 16-Byte aligned raises a misaligned memory access exception.
- As in non-AGU LD/ST operations, unaligned accesses from AGU based LD/ST are enabled by setting the AD bit in STATUS32, which disables alignment checking.
- AGUs can be used as operands in a PREFETCH instruction. In PREFETCH instructions an illegal AGU operand triggers an illegal-operand exception; illegal memory references do not trigger an exception as in non-AGU PREFETCH instructions.
- An AGU load and store operation can only be performed on the DCCM or data cache. AGU load and store operations cannot be performed on uncached data.

41.3 AGU Auxiliary Register Set

Table 41-1 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 0x5EF	AGU Modifier Registers, AGU_AUX_MODx	rw	Address generation unit modifier registers

41.3.1 AGU Address Pointer Registers, AGU_AUX_APx

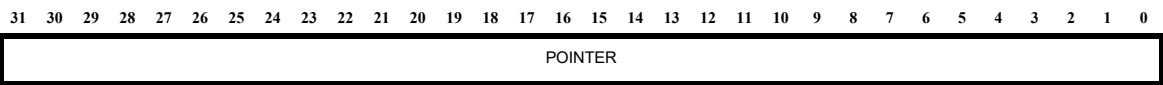
-

Address: 0x5C0 to 0x5CF

Access: rw

Reset: 0x0000_0000

Figure 41-1 AGU_AUX_APx Register



These registers are used as byte address pointers to the cache 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 41-2 Number of Address Pointer Registers

AGU Configuration (AGU_SIZE)	Number of Address Pointer Registers
small	4

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 cache or DCCM with the following boundaries:
 - lower boundary==base address of the memory region
 - upper boundary==lower boundary+wrap value

When the address pointer is incremented with the offset (register or immediate), it has to remain in the increment region. If the address pointer post increment crosses the upper boundary, the address

pointer wraps to the lower boundary. If the address pointer post increment crosses the lower boundary, the address pointer wraps to the upper boundary.

**Note**

When you use an AGU register both as a source and a destination operand, the modifier register is updated only once.

For a conditional instruction, the instruction is executed only if the condition is true.

If the condition is false, the instruction is skipped, the address-pointer registers are not updated, are no memory loads occur; both the source and destination addresses retain their values from before the update.

Table 41-3 Post-Incrementing Address Pointer Modes

AGU_AUX_MODx.[31:29] Value	Description
0	Linear increment with the signed offset from the register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field.
1	Linear increment with the unsigned immediate offset; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field.
2	Bit reverse increment with offset from register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field.
3	Bit reverse increment with immediate offset; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the direction to increment is specified in the AGU_AUX_MODx.[11] field; and the offset scaling factor is set by the AGU_AUX_MODx.[15:14] field.
4	Modulo wrapping increment with offset from register and modulo boundary from register; the offset register to use is specified in the AGU_AUX_MODx.[28:25] field; the wrap register to use is specified in the AGU_AUX_MODx.[17:14] field; and the direction to increment is specified in the AGU_AUX_MODx.[11] field.
5	Power of 2 modulo wrapping increment with immediate offset and immediate power of 2 modulo boundary; the immediate value is specified in the AGU_AUX_MODx.[28:18] field; the wrap immediate power is specified in the AGU_AUX_MODx.[18:14] field; and the direction to increment is specified in the AGU_AUX_MODx.[11] field.
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.

Table 41-3 Post-Incrementing Address Pointer Modes (Continued)

AGU_AUX_MODx.[31:29] Value	Description
7	When AGU_AUX_MODx.SUB_OP==0, update the modifier specified by the SEC_PTR_REG with the address from the AGU_AUX_MODx.[3:0] field and the AGU_AUX_MODx.[28:25] field. 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:

**Note**

- The address-pointer value is updated only if the offset is non-zero or if AGU_AUX_MODx.[31:29]==7.
- If multiple instruction operands refer to the same address pointer, the zero offset update is ignored unless AGU_AUX_MODx.[31:29]==7.
- If multiple instruction operands refer to the same address pointer using non-zero offset values or with AGU_AUX_MODx.[31:29]==7, the behavior is undefined.

If insufficient modifiers are available in the hardware configuration, you can use the MODIF instruction explicitly to update the address pointer.

Table 41-4 AGU_AUX_APx Field Description

Field	Bit	Description
POINTER	[31:0]	Byte address pointers to values in the cache or DCCM memories

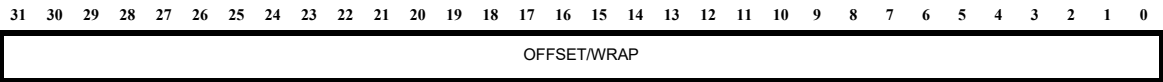
41.3.2 AGU Offset Registers, AGU_AUX_OSx

Address: 0x5D0 to 0x5DF

Access: rw

Reset: 0x0000_0000

Figure 41-2 AGU_AUX_OSx Register



These registers are used to determine the increment values and wrap boundaries during the address generation process. If used as an offset value, the value is the non-scaled value that is added to or subtracted from the address pointer. If used as a wrap boundary, this value should specify the size of the modulo buffer in bytes minus one.

The width of the register depends on the ADDR_SIZE configuration option. If ADDR_SIZE < 32, the most significant bits [31: (32-ADDR_SIZE)] are read as zero.

The number of offset or wrap registers included in the processor is based on the AGU configuration.

Table 41-5 Number of Offset or Wrap Registers

AGU Configuration (AGU_SIZE)	Number of Offset or Wrap Registers
small	2

After a memory access, the pointer value in the address pointer register is incremented based on the rules in [Table 41-3](#) on page 1446.

41.3.3 AGU Modifier Registers, AGU_AUX_MODx

Address: 0x5E0 to 0x5F7

Access: rw

Reset: 0x0000_0000

The modifier registers define the type of address arithmetic to be performed during address generation and also the data transformations performed on the data vectors. The modifier registers allow you to create data structures in memory. The format of the modifier registers depends on the addressing mode and the data transformation to be performed.

Table 41-6 Number of Modifier Registers

AGU Configuration (AGU_SIZE)	Number of Modifier Registers
small	4 (addresses 0x5E0 to 0x5E3 are used within the AGU modified register range (0x5E to 0x5F7, the unused registers are reserved).

Figure 41-3 AGU_AUX_MODx Register with Offset Register Increment

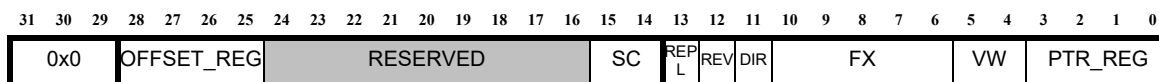


Figure 41-4 AGU_AUX_MODx Register with Immediate Offset

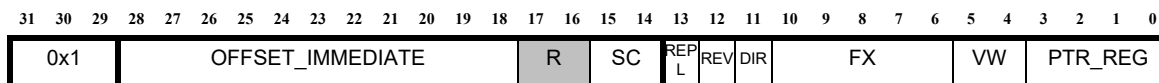


Figure 41-5 AGU_AUX_MODx Register with Bit Reverse Increment Using Offset Register

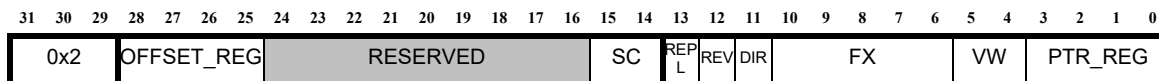


Figure 41-6 AGU_AUX_MODx Register with Bit Reverse Increment Using Immediate Offset

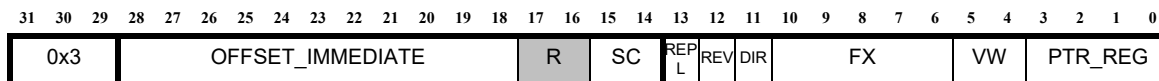


Figure 41-7 AGU_AUX_MODx Register with Modulo Wrapping Using Offset and Wrap Registers

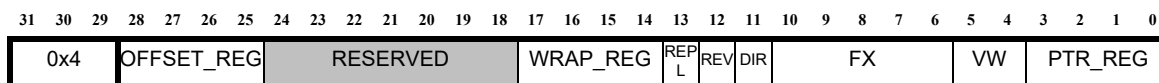


Figure 41-8 AGU_AUX_MODx Register with Modulo Wrapping Using Immediate Offset and Wrap

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
0x5				OFFSET_IMMEDIATE										WRAP_IMM			REP L	REV	DIR	FX				VW		PTR_REG					

Figure 41-9 AGU_AUX_MODx with Modulo Wrapping Using Offset Register and Immediate Wrap

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
0x6				OFFSET_REG				RESERVED							WRAP_IMM			REP L	REV	DIR	FX				VW		PTR_REG				

Figure 41-10 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset 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
0x7				OFFSET_REG				0x0				RESERVED				SEC_PTR_REG			REP L	REV	DIR	FX				VW		PTR_REG			

Figure 41-11 AGU_AUX_MODx Register when Performing Pointer Arithmetic with Offset 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	
0x7				RESERVED				0x1				OFFSET_IMM				SEC_PTR_REG				REP L	REV	DIR	FX				VW		PTR_REG			

Field	Bit	Description
PTR_REG	[3:0]	Indicates the address pointer register index
DIR	11	Indicates if the address pointer should be incremented or decremented. 1: decrement the address pointer 0: increment the address pointer
SC	[15:14]	Indicates the offset scaling factor. effective_offset = offset * 2 ^{SC}
WRAP_IMM	[17:14]	Specifies the unsigned immediate wrap value; this value is used to create a region within the memory with the following boundaries: - lower_boundary == base address of the memory region - upper_boundary == lower_boundary + 2^{WRAP_IMM+1} The address pointer has to remain within this region. If the address pointer goes out of bounds of this region, the address pointer wraps to the lower or the upper boundary of the region. This field is available as a wrap immediate only if 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. upper_boundary == lower_boundary + offset[WRAP_REG]
SEC_PTR_REG	[17:14]	Specifies a destination AGU Address Pointer Register (AGU_AUX_APx) defining the address pointer to be used for the address update
SUB_OP	[24:22]	This field is specific to OPC==7. Specifies whether an offset register or offset immediate is used in the pointer arithmetic.

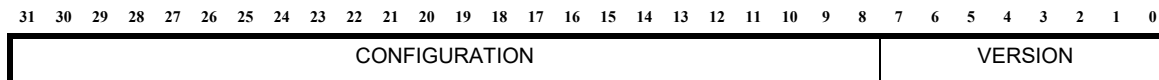
Field	Bit	Description
OFFSET_IMM	[28:18]	Specifies the unsigned immediate offset value by which the address pointer is incremented or decremented. This field is available only if <code>OPC==1 3 6 7</code> . When <code>OPC=7</code> , the <code>OFFSET_IMM</code> field is at position [21:19], the bits [24:22] are taken up by the <code>SUB_OP</code> field, and [28:25] are reserved.
OFFSET_REG	[28:25]	Specifies the index of the offset register that holds the signed offset value. This field is available only if <code>OPC==0 2 4 5 7</code> .
OPC	[31:29]	This field controls the arithmetic to increment or decrement the address pointer. 0: Linear increment with signed offset from register; the offset register to use is specified in the <code>AGU_AUX_MODx</code>. [28:25] field; the direction to increment is specified in the <code>AGU_AUX_MODx</code>. [11] field; and the offset scaling factor is set by the <code>AGU_AUX_MODx</code>. [15:14] field. 1: Linear increment with unsigned immediate offset; the immediate value is specified in the <code>AGU_AUX_MODx</code>. [28:18] field; the direction to increment is specified in the <code>AGU_AUX_MODx</code>. [11] field; and the offset scaling factor is set by the <code>AGU_AUX_MODx</code>. [15:14] field. 2: Bit reverse increment with offset from register; the offset register to use is specified in the <code>AGU_AUX_MODx</code>. [28:25] field; the direction to increment is specified in the <code>AGU_AUX_MODx</code>. [11] field; and the offset scaling factor is set by the <code>AGU_AUX_MODx</code>. [15:14] field. 3: Bit reverse increment with immediate offset; the immediate value is specified in the <code>AGU_AUX_MODx</code>. [28:18] field; the direction to increment is specified in the <code>AGU_AUX_MODx</code>. [11] field; and the offset scaling factor is set by the <code>AGU_AUX_MODx</code>. [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 <code>AGU_AUX_MODx</code>. [28:25] field; the wrap register to use is specified in the <code>AGU_AUX_MODx</code>. [17:14] field; and the direction to increment is specified in the <code>AGU_AUX_MODx</code>. [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 <code>AGU_AUX_MODx</code>. [28:18] field; the wrap immediate power is specified in the <code>AGU_AUX_MODx</code>. [18:14] field; and the direction to increment is specified in the <code>AGU_AUX_MODx</code>. [11] field. 6: Modulo wrapping increment with offset from register and immediate power of 2 modulo boundary; the offset register to use is specified in the <code>AGU_AUX_MODx</code>. [28:25] field; the wrap immediate power is specified in the <code>AGU_AUX_MODx</code>. [18:14] field; and the direction to increment is specified in the <code>AGU_AUX_MODx</code>. [11] field. 7: When <code>SUB_OP==0</code>, update the modifier specified by the <code>SEC_PTR_REG</code> with the address from the <code>AGU_AUX_MODx</code>. [3:0] field and the <code>AGU_AUX_MODx</code>. [28:25] field. 7: When <code>SUB_OP==1</code>, update the modifier specified by the <code>SEC_PTR_REG</code> with the address from the <code>AGU_AUX_MODx</code>. [3:0] register and the immediate offset specified by the <code>AGU_AUX_MODx</code>. [21:18] field.

41.3.4 AGU Build Configuration Register, AGU_BUILD

Address: 0xCC

Access: R

Figure 41-12 AGU_BUILD Register



This register indicates the presence of the Address Generator Unit (AGU) and its configuration.

Table 41-7 AGU_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of the AGU unit 0x03: first version for ARC HS.
CONFIGURATION	[11:8]	Number of address pointers The number of address pointers is defined by the build option <code>agu_size</code> . If <code>agu_size==small</code>; number of address pointers = 4; this field is fixed to 4.
	[15:12]	Number of offset registers The number of offset registers is defined by the build option <code>agu_size</code> . If <code>agu_size==small</code>; number of offset registers = 2; this field is fixed to 2.
	[20:16]	Number of modifiers The number of modifier registers is defined by the build option <code>agu_size</code> . If <code>agu_size==small</code>; number of modifier registers= 4; this field is fixed to 4.
	[23:21]	This field is fixed to 0.
	[24]	This field is fixed to 0.
	[25]	Indicates the presence of the simple AGU scheme This field is fixed to 0.
	[31:26]	Reserved

42

Scalar DSP Instruction Set Details

42.1 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). . .
- 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.

42.2 List of Instructions

Table 42-1 summarizes the 32-bit alongside the 16-bit instructions supported by the ARCV2 ISA.

Table 42-1 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
ASRSR	Shift right rounding and saturating if value is negative

Table 42-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
CBFLYHF0R	16 bit + 16 bit FFT butterfly, first half; return first result
CBFLYHF1R	16 bit + 16 bit FFT butterfly, second half; return first result with rounding
CMACCHFR	16 bit + 16 bit complex signed fractional conjugated multiplication and accumulation with rounding
CMACCHNFR	16 bit + 16 bit complex signed fractional conjugated multiplication and accumulation with rounding, and returned without a fractional shift
CMACHFR	16 + 16 bit complex signed fractional multiplication and accumulation with rounding
CMACHNFR	16 bit + 16 bit complex signed fractional multiplication and accumulation with rounding
CMPYCHFR	16 bit + 16 bit complex signed fractional conjugated multiplication with rounding
CMPYCHNFR	16 bit + 16 bit complex signed fractional conjugated multiplication with rounding, and returned without a fractional shift
CMPYHFMR	16 + 16 bit complex signed fractional multiplication with rounding
CMPYHNFR	16 bit + 16 bit complex signed fractional multiplication with rounding
DIV	2's complement integer division
DIVF	Signed 32-bit fractional division
DIVU	Unsigned integer Divide
DMACH	Dual 16x16 multiply and accumulate
DMACHBL	Dual vector 16 bitx8 bit signed integer multiplication and accumulation; lower bytes
DMACHBM	Dual vector 16 bitx8 bit signed integer multiplication and accumulation; higher bytes
DMACHF	Dual vector 16 bit signed fractional multiplication and accumulation
DMACHFR	Dual vector 16 bit signed fractional rounding multiplication and accumulation
DMACHU	Dual unsigned 16x16 multiply and accumulate
DMACWH	Dual 32x16 multiply and accumulate
DMACWHF	Dual 32x16 fractional multiply and accumulate
DMACWHU	Dual unsigned 32x16 multiply and accumulate
DMPYH	Dual 16x16 multiplication
DMPYHBL	Dual vector 16 bitx8 bit signed integer multiplication; lower bytes
DMPYHBM	Dual vector 16 bitx8 bit signed integer multiplication; higher bytes

Table 42-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
DMPYHF	Dual vector 16 bit signed fractional multiplication
DMPYHFR	Dual vector 16 bit signed fractional rounding multiplication
DMPYHU	Dual unsigned 16x16 multiplication
DMPYHWF	Dual vector 16 bit signed fractional multiplication and 32-bit writeback
DMPYWH	Sum of dual 32x16 multiplication
DMPYWHF	Dual signed 32x16 fractional multiply and accumulate
DMPYWHU	Sum of dual unsigned 32x16 multiplication
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.

Table 42-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
MPY	32 x 32 Signed Multiply (lsw)
MPY_S	Signed 32 x 32 multiplication. Return the least-significant 32 bits of the result.
MPYM MPYH	32 x 32 Signed Multiply (msw)
MPYD	32x32 multiplication, double result
MPYDF	32 bit signed fractional saturating multiplication; 64-bit result
MPYDU	Unsigned 32x32 multiplication, double result
MPYF	32 bit signed fractional saturating multiplication
MPYFR	32 bit signed fractional saturating an rounding multiplication
MPYMU MPYHU	32 x 32 Unsigned Multiply (msw)
MPYU	32 x 32 Unsigned Multiply (lsw)
MPYUW	16 bit unsigned integer multiplication
MPYUW_S	16 bit unsigned integer multiplication
MPYW	16 X 16 signed multiply
MPYW_S	16 X 16 signed multiply
MPYWHFL	32x16 bit signed fractional multiplication, lower signed 16 bits
MPYWHFLR	32x16 bit signed fractional multiplication with rounding, lower signed 16 bits
MPYWHFM	32x16 bit signed fractional multiplication, higher signed 16 bits
MPYWHFMR	32x16 bit signed fractional multiplication with rounding, higher signed 16 bits
MPYWHL	32x16 bit signed multiplication, lower signed 16 bits
MPYWHKL	Signed 32x16 integer multiplication, and shift-right the lower bits
MPYWHKUL	Unsigned 32x16 integer multiplication, and shift-right the lower bits
MPYWHUL	32x16 bit unsigned multiplication, lower signed 16 bits
MSUBDF	32 bit signed fractional saturating subtraction; 64-bit result
MSUBFR	32 bit fractional saturating and rounding subtraction
MSUBF	32 bit fractional saturating subtraction
MSUBWHFL	Signed 32x16 fractional multiplication and subtraction
MSUBWHFLR	Signed 32x16 fractional multiplication and subtraction with rounded result

Table 42-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
MSUBWHFM	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand
MSUBWHFMR	Signed fractional multiplication and subtraction of 32 bits of the b operand and the most-significant 16 bits of the c operand with rounded result
NEGS	Negative saturated 16-bit value
NEGSH	Negative saturated 32-bit value
NORMACC	Normalize the accumulator
QMACH	Quad 16x16 multiply and accumulate
QMACHF	Quad 16x16 signed fractional multiply and accumulate; result is saturated
QMACHU	Quad unsigned 16x16 multiply and accumulate
QMPYH	Quad 16x16 multiplication
QMPYHF	Quad 16x16 fractional multiplication; return a 16-bit saturated fractional result
QMPYHU	Quad unsigned 16x16 multiply
REM	Signed Integer Remainder
REMU	Unsigned Integer Remainder
RNDH	Rounding and saturating a 32-bit value to a 16-bit value
SATH	Saturating 32-bit value to a 16-bit value
SATF	Saturate a register based on the STATUS32.N and STATUS32.V flags
SBCS	Signed subtraction with saturation and carry in
SBCS	Subtract with carry
SETACC	Set the accumulator
SQRT	Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result
SQRTF	Compute the square-root of an unsigned Q31 fraction, yielding a Q31 fractional result
SUBS	Saturating 32-bit subtraction
VABS2H	Two way 16 bit vector absolute
VABSS2H	Two way 16 bit vector saturating absolute
VADD2	Dual 32-bit vector addition
VADD2H	Dual 16-bit vector addition

Table 42-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
VADD4H	Quad 16-bit vector addition
VADD4B	Four way add
VADDS2	Two-way 32-bit vector addition
VADDS2H	Two way vector saturating add 16 bits
VADDS4H	Two-way 32-bit vector subtraction. The result is saturated.
VADDSUB2H	Dual 16-bit vector add and subtract
VADDSUB	Two-way 32-bit vector addition and subtraction.
VADDSUBS	Two-way 32-bit vector addition and subtraction. The results are saturated and stored.
VADDSUBS2H	Two way vector saturating add subtract
VADDSUB4H	Quad 16-bit SIMD add and subtract.
VADDSUBS4H	Four-way 16-bit vector addition and subtraction. The results are saturated and stored.
VALGN2H	Two-way 16-bit vector align
VASL2H	Two way arithmetic shift left 16 bits
VASLS2H	Two way saturating arithmetic shift left 16 bits
VASR2H	Two way arithmetic shift right 16 bits
VASRS2H	Two way arithmetic saturating shift right 16 bits
VASRSR2H	Two way arithmetic saturating and rounding shift right 16 bits
VEXT2BHL	Vector extend the lower two bytes to the higher two bytes
VEXT2BHL	Compose a dual Q15 vector by expanding the lower two Q7 bytes of an operand
VEXT2BHM	Vector extend the higher two bytes to the lower two bytes
VEXT2BHMF	Compose a dual Q15 vector by expanding the higher two Q7 bytes of an operand
VLSR2H	Two way vector logic shift right 16 bits
VMAC2H	Dual 16x16 vector multiplication and accumulation
VMAC2HF	Two way vector 16 bit saturating fractional multiplication and accumulation
VMAC2HFR	Two way vector 16 bit saturating and rounding fractional multiplication and accumulation
VMAC2HNFR	Two way vector 16 bit saturating and rounding fractional multiplication and accumulation

Table 42-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
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
VPACK2HBMF	Pack the higher two bytes of a vector from the higher 8-bits of two 16-bit vectors of an operand
VPERM	Perform byte permutation with zero or sign extension
VREP2HL	Replicate lower 16-bits into the upper 16-bits
VREP2HM	Replicate higher 16-bits to lower 16-bits
VSEXT2BHL	Vector sign extend the lower two bytes to the higher two bytes

Table 42-1 List of Instructions (Continued)

32-Bit Instructions	
Instruction	Operation
VSEXT2BHM	Vector sign extend the higher two bytes to the lower two bytes
VSUB2	Dual 32-bit SIMD subtract.
VSUB4B	Four way subtract
VSUBS4H	Four-way 16-bit vector addition.
VSUB2H	Dual 16-bit vector subtraction
VSUB4H	Quad 16-bit SIMD subtraction
VSUBS2	Two-way 32-bit vector subtraction. The result is saturated.
VSUBADD2H	Dual 16-bit vector subtract and add
VSUBADDS	Two-way 32-bit vector subtraction and addition. The results are saturated
VSUBADDS4H	Four-way 16-bit vector subtraction and addition. The results are saturated.
VSUBADDS2H	Two way vector saturating subtract add
VSUBS2H	Two way vector saturating subtract 16 bits

ABSS

Function

Compute the absolute value of a 32-bit operand and saturate the result.

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT32}(\text{ABS}(c)) ;$$

Instruction Format

op b, c

Syntax Example

ABSS<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of the input is set to 1
C		= Unchanged
V	•	= Set if src = 0x8000 0000; otherwise cleared

 **Note**

The value of the input operands is used to set the flags.

Description

Compute the absolute value of a 32-bit operand, *c*.. Saturate the result and store in operand *b*. The absolute value of 0x8000_0000 yields 0x7FFF_FFFF. The saturation flag DSP_CTRL.SAT is set, regardless of the <.F> suffix, if the instruction saturates. If the set flags suffix (.F) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = c >= 0 ? c : (c == 0x8000_0000 ? 0x7fff_ffff : -c);
DSP_CTRL.SAT |= c == 0x8000_0000;
if (F) {
    STATUS32.Z = c == 0x0000_0000;
    STATUS32.N = c < 0;
    STATUS32.V = c == 0x8000_0000;
}
```

Assembly Code Example

```
ABSS r1,r2    ; Take the absolute saturated value of r2 and write
               ;result into r1
```

Syntax and Encoding

Instruction Code

ABSS<.f>	b, c	00101 bbb 00 101111 FBBB CCCCC 000101
ABSS<.f>	b, limm	00101 bbb 00 101111 FBBB 111110 000101
ABSS<.f>	0, c	00101 110 00 101111 F111 CCCCC 000101
ABSS<.f>	0, limm	00101 110 00 101111 F111 111110 000101
ABSS<.f>	b, u6	00101 bbb 01 101111 FBBB uuuuuu 000101
ABSS<.f>	0, u6	00101 110 01 101111 F111 uuuuuu 000101

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ABSSH

Function

Compute the absolute value of a 16-bit element and saturate the result.

Extension Group

DSP basic arithmetic

Operation

```
b = SAT16 (ABS (c.h0)) ;
```

Instruction Format

op b, c

Syntax Example

```
ABSSH<.f> b,c
```

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of the input is set to 1
C		= Unchanged
V	•	= Set if src = 0x8000; otherwise cleared



Note

The value of the input operands is used to set the flags. The flag setting is in relation to a 16-bit operand (and not a 32-bit operand as with other instructions).

Description

Compute the absolute value of a 16-bit operand, c,. Saturate the result and store in operand b. The absolute value of 0x8000 yields 0x7FFF. 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

```
t = c & 0x0000_ffff;
b = t <= 0x7fff ? t : (t == 0x8000 ? 0x7fff : 0x10000-t);
DSP_CTRL.SAT |= t == 0x8000;
if (F) {
    STATUS32.Z = t == 0x0000;
    STATUS32.N = t >= 0x8000;
    STATUS32.V = t == 0x8000;
}
```

Assembly Code Example

```
ABSSH r1,r2    ; Take the absolute saturated value of the lower 16 bits
                ; of r2 and write result into r1
```

Syntax and Encoding

Instruction Code

ABSSH<.f>	b,c	00101 bbb 00 101111 FBBB CCCCC 000100
ABSSH<.f>	b,limm	00101 bbb 00 101111 FBBB 111110 000100
ABSSH<.f>	0,c	00101 110 00 101111 F111 CCCCC 000100
ABSSH<.f>	0,limm	00101 110 00 101111 F111 111110 000100
ABSSH<.f>	b,u6	00101 bbb 01 101111 FBBB uuuuuu 000100
ABSSH<.f>	0,u6	00101 110 01 101111 F111 uuuuuu 000100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ADCS

Function

Signed addition with saturation and carry in

Extension Group

DSP basic arithmetic

Operation

$$a = \text{SAT32}(b + c + \text{STATUS32}.C)$$

Instruction Format

op a, b, c

Syntax Example

ADCS<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Cleared
V	•	= Set if the result is saturated

Description

Add source operand 1 (b) and source operand 2 (c) and the carry flag (STATUS32.C); store the saturated result in the destination register, a. If the addition overflows, the result is saturated to the maximum positive value 0x7FFF_FFFF or the minimum negative value 0x8000_0000.

The saturation flag, DSP_CTRL.SAT, is set if the result of the instruction saturates, regardless of the .F suffix.

Pseudo Code

```
s = sat((sint128) b + c + STATUS32.C, 31, &a);    /* ADCS */
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
    STATUS32.C = 0;
}
```

Assembly Code Example

```
ADCS r0,r1,r2    ; Add r2 to r1 with carry and write the saturated
                  ; result to r0
```

Syntax and Encoding

		Instruction Code
ADCS<.f>	a,b,c	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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ADDS

Function

Signed 32-bit addition. The result is saturated.

Extension Group

DSP basic arithmetic

Operation

$$a = \text{SAT32} (b + c)$$

Instruction Format

op a, b, c

Syntax Example

ADDS<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Cleared
V	•	= Set if the output is saturated

Description

Compute the sum of operands `b` and `c`. The result is stored in the destination register, `a`. If the addition overflows, the result is saturated to the maximum positive value `0x7fff_ffff` or the minimum negative value `0x8000_0000`. The saturation flag `DSP_CTRL.SAT` is set if the result of the instruction saturates, regardless of the `.F` suffix. If the set flags suffix (`.F`) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
s = sat((sint128) b + c, 31, &a);          /* ADDS */
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
    STATUS32.C = 0;
}
```

Assembly Code Example

```
ADDS r0,r1,r2    ; Add contents of r1 with r2 and write result into r0
                  ; while saturating
```

Syntax and Encoding

Instruction Code

ADDS<.f>	a, b, c	00101bbb00000110FBBBCCCCC	AAAAAA
ADDS<.f>	a, b, u6	00101bbb01000110FBBBuuuuuu	AAAAAA
ADDS<.f>	b, b, s12	00101bbb10000110FBBBssssss	SSSSSS
ADDS<.cc><.f>	b, b, c	00101bbb11000110FBBBCCCCC	0QQQQQ
ADDS<.cc><.f>	b, b, u6	00101bbb11000110FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ASLACC

Function

Perform an arithmetic shift of the accumulator.

Extension Group

DSP Accumulator

Operation

```
switch (c.b1) {
0: accwide = accwide << c.b0;
1: acclo = acclo << c.b0;
2: acchi = acchi << c.b0;
3: acchi = acchi << c.b2; acclo = acclo << c.b0;
}
```

Instruction Format

op c

Syntax Example

ASLACC c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an arithmetic left shift of the accumulator, including guard bits. The `c` operand consists of two fields: the shift amount (`c.b0`) and the accumulator select (`c.b1`). The shift amount is a signed integer; a negative value effectively shifts the accumulator right, without rounding.

Using the loop-counter (LP_COUNT/r60) as a source operand to the ASLACC instruction results in an illegal instruction exception.

If, `dsp_accshift==full`, the shift amount is saturated to a range of $-72 \leq \text{shamt} \leq +72$; if `dsp_accshift==limited`, the shift amount is saturated to a range of $-8 \leq \text{shamt} \leq +8$.

Operand Attributes

Operand c.b1	Description
0	selects the wide accumulator
1	selects the low accumulator
2	selects the high accumulator
3	selects dual hi/lo accumulator output; move c.b2 into the high accumulator, and move c.b0 into the low accumulator

Pseudo Code

```

if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)                                     // ASLACC
    shamt = c.b0 < -72 ? -72 : (c.b0 > 72 ? 72 : c.b0);
else
    shamt = c.b0 < -8 ? -8 : (c.b0 > 8 ? 8 : c.b0);
switch (c.b1 & 3) {
    case 0:
        if (DSP_CTRL.GE)
            accwide = ACC0_GHI.GUARD << 64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        accwide = (shamt >= 0) ? accwide << shamt : accwide >> -shamt;
        if (DSP_CTRL.GE)
            ACC0_GHI.GUARD = accwide >> 64 && 0xff;
            ACC0_HI = accwide >> 32 && 0xffff_ffff;
            ACC0_LO = accwide && 0xffff_ffff;
            break;
    case 1:
        if (DSP_CTRL.GE)
            acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
        else
            acclo = ACC0_LO;
        acclo = (shamt >= 0) ? acclo << shamt : acclo >> -shamt;
        if (DSP_CTRL.GE)
            ACC0_GLO.GUARD = acclo >> 32 && 0xff;
            ACC0_LO = acclo && 0xffff_ffff;
            break;
    case 2:
        if (DSP_CTRL.GE)
            acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        else
            acchi = ACC0_HI;
        acchi = (shamt >= 0) ? acchi << shamt : acchi >> -shamt;
        if (DSP_CTRL.GE)
            ACC0_GHI.GUARD = acchi >> 32 && 0xff;
            ACC0_HI = acchi && 0xffff_ffff;
            break;
}

```

```
case 3:
    shamtl = sext(c.b0);
    shamth = sext(c.b2);
    if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)
        shamtl = shamtl < -72 ? -72 : (shamtl > 72 ? 72 : shamtl);
        shamth = shamth < -72 ? -72 : (shamth > 72 ? 72 : shamth);
    else
        shamtl = shamtl < -8 ? -8 : (shamtl > 8 ? 8 : shamtl);
        shamth = shamth < -8 ? -8 : (shamth > 8 ? 8 : shamth);
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
    else
        acchi = ACC0_HI;
        acclo = ACC0_LO;
    acchi = (shamth >= 0) ? acchi << shamth : acchi >> -shamth;
    acclo = (shamtl >= 0) ? acclo << shamtl : acclo >> -shamtl;
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = acchi >> 32 && 0xff;
        ACC0_GLO.GUARD = acclo >> 32 && 0xff;
    ACC0_HI = acchi && 0xffff_ffff;
    ACC0_LO = acclo && 0xffff_ffff;
    break;
}
```

Assembly Code Example

```
ASLACC r2          ; Shift left the accumulator based on fields in r2
```

Syntax and Encoding

Instruction Code		
ASLACC	c	00101000001011110000CCCCC111111
ASLACC	u6	00101000011011110000uuuuuu111111

ASLS

Function

Perform an arithmetic shift left operation on an operand. The shift amount is specified in the second operand. Store the saturated result in the destination register.

Extension Group

DSP basic arithmetic

Operation

$$a = c >= 0 ? \text{SAT32}(b << c) : b >> -c$$

Instruction Format

op a, b, c

Syntax Example

ASLS<.f> a, b, c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if the result is saturated

Description

If operand *c* is positive, perform an arithmetic shift left on operand *b* by the shift amount is specified by in the *c* operand. Return the saturated result. If operand *c* is negative, perform a shift right by $-c$ bits. The left-shift operation saturates if any of the bits shifted out is unequal to the original sign bit or if the result sign bit is different from the original sign bit.

The shift amount is saturated to $[-31, 32]$.

The saturation flag `DSP_CTRL.SAT` is set if the result of the instruction saturates regardless of the `.F` suffix. Other flag updates only occur if the set flags suffix (`.F`) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

if (c >= 0) {
    c = c > 32 ? 32 : c;
    s = sat((sint128) b << c, 31, &a);
} else {
    s = 0;
    a = b >> (c < -31 ? 31 : -c);
}
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
}

```

Assembly Code Example

```

ASLS r0,r1,r2      ; Arithmetic shift left r1 by r2 bit positions and write
                   ;result into r0 while saturating

```

Syntax and Encoding

Instruction Code

ASLS<.f>	a,b,c	00101 bbb 00 001010 F BBB CCCCC AAAAAA
ASLS<.f>	a,b,u6	00101 bbb 01 001010 F BBB uuuuuu AAAAAA
ASLS<.f>	b,b,s12	00101 bbb 10 001010 F BBB ssssss SSSSSS
ASLS<.f><.cc>	b,b,c	00101 bbb 11 001010 F BBB CCCCC 0QQQQQ
ASLS<.f><.cc>	b,b,u6	00101 bbb 11 001010 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ASLSACC

Function

Perform an arithmetic shift left operation on the accumulator. Store the saturated result.

Extension Group

DSP Accumulator

Operation

```
if (DSP_CTRL.GE) {
    switch (c.b1) {
        0: accwide = SAT72(accwide << c.b0);
        1: acclo = SAT40(acclo << c.b0);
        2: acchi = SAT40(acchi << c.b0);
        3: acchi = SAT40(acchi << c.b2); acclo = SAT40(acclo << c.b0);
    } else {
        switch (c.b1) {
            0: accwide = SAT64(accwide << c.b0);
            1: acclo = SAT32(acclo << c.b0);
            2: acchi = SAT32(acchi << c.b0);
            3: acchi = SAT32(acchi << c.b2); acclo = SAT32(acclo << c.b0);
        }
    }
}
```

Instruction Format

op c

Syntax Example

ASLSACC c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an arithmetic shift left operation on the accumulator including the guard bits. Store the saturated result. The c operand consists of two fields: the shift amount and the accumulator select.

The shift amount is a signed integer; a negative value effectively shifts the accumulator right, with saturation and without rounding. The sticky flag, `DSP_CTRL.SAT`, is set if the result saturates.

If, `dsp_accshift==full`, the shift amount is saturated to a range of $-72 \leq \text{shamt} \leq +72$; if `dsp_accshift==limited`, the shift amount is saturated to a range of $-8 \leq \text{shamt} \leq +8$.

Using the loop-counter (`LP_COUNT/r60`) as a source operand to the `ASLSACC` instruction results in an illegal instruction exception.

Operand Attributes

Operand c.b1	Description
0	selects the wide accumulator
1	selects the low accumulator
2	selects the high accumulator
3	selects dual hi/lo accumulator output; move c.b2 into the high accumulator, and move c.b0 into the low accumulator

Pseudo Code

```

if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)                                /* ASLSACC
    shamt = c.b0 < -72 ? -72 : (c.b0 > 72 ? 72 : c.b0);                */
else
    shamt = c.b0 < -8 ? -8 : (c.b0 > 8 ? 8 : c.b0);
switch (c.b1 & 3) {
    case 0:
        if (DSP_CTRL.GE)
            accwide = ACC0_GHI.GUARD << 64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            msb = 71;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            msb = 63;
        if (shamt >= 0)
            s = sat(accwide<<shamt, msb, &accwide); // requires 256b arith
        else
            s = 0;
            accwide = accwide >> -shamt;
        DSP_CTRL.SAT |= s;
        if (DSP_CTRL.GE)
            ACC0_GHI.GUARD = accwide >> 64 && 0xff;
            ACC0_HI = accwide >> 32 && 0xffff_ffff;
            ACC0_LO = accwide && 0xffff_ffff;
            break;
    case 1:
        if (DSP_CTRL.GE)
            acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
            msb = 39;
        else
            acclo = ACC0_LO;
            msb = 31;
        if (shamt >= 0)
            s = sat(acclo<<shamt, msb, &acclo); // requires 256b arith
        else
            s = 0;
            acclo = acclo >> -shamt;
        DSP_CTRL.SAT |= s;
        if (DSP_CTRL.GE)
            ACC0_GLO.GUARD = acclo >> 32 && 0xff;
            ACC0_LO = acclo && 0xffff_ffff;
            break;

```

```

case 2:
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        msb = 39;
    else
        acchi = ACC0_HI;
        msb = 31;
    if (shamt >= 0)
        s = sat(acchi<<shamt, msb, &acchi); // requires 256b arith
    else
        s = 0;
        acchi = acchi >> -shamt;
    DSP_CTRL.SAT |= s;
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = acchi >> 32 && 0xff;
    ACC0_HI = acchi && 0xffff_ffff;
    break;
case 3:
    shamtl = sext(c.b0);
    shamth = sext(c.b2);
    if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)
        shamtl = shamtl < -72 ? -72 : (shamtl > 72 ? 72 : shamtl);
        shamth = shamth < -72 ? -72 : (shamth > 72 ? 72 : shamth);
    else
        shamtl = shamtl < -8 ? -8 : (shamtl > 8 ? 8 : shamtl);
        shamth = shamth < -8 ? -8 : (shamth > 8 ? 8 : shamth);
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
        msb = 39;
    else
        acchi = ACC0_HI;
        acclo = ACC0_LO;
        msb = 31;
    if (shamtl >= 0)
        sl = sat(acclo<<shamtl, msb, &acclo); // requires 256b arith
    else
        sl = 0;
        acclo = acclo >> -shamtl;
    if (shamth >= 0)
        sh = sat(acchi<<shamth, msb, &acchi); // requires 256b arith
    else
        sh = 0;
        acchi = acchi >> -shamth;
    DSP_CTRL.SAT |= sl || sh;
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = acchi >> 32 && 0xff;
        ACC0_GLO.GUARD = acclo >> 32 && 0xff;
    ACC0_HI = acchi && 0xffff_ffff;
    ACC0_LO = acclo && 0xffff_ffff;
    break;
}

```

```

case 3:
    shamtl = sext(c.b0);
    shamth = sext(c.b2);
    shamtl = shamtl < -64 ? -64 : (shamtl > 63 ? 63 : shamtl);
    shamth = shamth < -64 ? -64 : (shamth > 63 ? 63 : shamth);
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
        msb = 39;
    else
        acchi = ACC0_HI;
        acclo = ACC0_LO;
        msb = 31;
    if (shamtl >= 0)
        sl = sat(acclo<<shamtl, msb, &acclo); // requires 256b arith
    else
        sl = 0;
        acclo = acclo >> -shamtl;
    if (shamth >= 0)
        sh = sat(acchi<<shamth, msb, &acchi); // requires 256b arith
    else
        sh = 0;
        acchi = acchi >> -shamth;
    DSP_CTRL.SAT |= sl || sh;
    if (DSP_CTRL.GE)
        ACC0_GHI.GUARD = acchi >> 32 && 0xff;
        ACC0_GLO.GUARD = acclo >> 32 && 0xff;
    ACC0_HI = acchi && 0xffff_ffff;
    ACC0_LO = acclo && 0xffff_ffff;
    break;
}

```

Assembly Code Example

```

ASLSACC r2      ; Shift left the accumulator and saturate based on fields
                 ;in r2

```

Syntax and Encoding

Instruction Code

ASLSACC	c	00101001001011110000CCCCC111111
ASLSACC	u6	00101001011011110000uuuuuu111111

ASRS

Function

Perform an arithmetic shift right operation on an operand. The shift amount is specified in the second operand. Store the saturated result in the destination register.

Extension Group

DSP basic arithmetic

Operation

$$a = c \gg 0 ? b \gg c : \text{SAT32}(b \ll -c) ;$$

Instruction Format

op a, b, c

Syntax Example

ASRS<.f> a, b, c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if the result is saturated

Description

Perform an arithmetic shift right operation on an operand. The shift amount is specified in the *c* operand. Store the saturated result in the destination register. If operand *c* is negative, perform an arithmetic shift left—shift amount is specified in the *c* operand and return the saturated result. The left shift saturates if any of the bits shifted out is unequal to the original sign bit or if the result sign bit is different from the original sign bit.

The shift amount is saturated to [-32,31].

The saturation flag `DSP_CTRL.SAT` is set if the result of the instruction saturates regardless of the `.F` suffix. Other flag updates only occur if the set flags suffix (`.F`) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

if (c >= 0) {
    s = 0;
    a = b >> (c > 31 ? 31 : c);
} else {
    c = c < -32 ? -32 : c;
    s = sat((sint128) b << -c, 31, &a);
}
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
}

```

Assembly Code Example

```

ASRS r0,r1,r2    ; Arithmetic shift right r1 by r2 bit positions and write
                  ;result into r0 while saturating

```

Syntax and Encoding

Instruction Code

ASRS<.f>	a,b,c	00101 bbb 00001011FBBBCCCCC AAAAAA
ASRS<.f>	a,b,u6	00101 bbb 01001011FBBBuuuuuu AAAAAA
ASRS<.f>	b,b,s12	00101 bbb 10001011FBBBssssss SSSSSS
ASRS<.cc><.f>	b,b,c	00101 bbb 11001011FBBBCCCCC 0QQQQQ
ASRS<.cc><.f>	b,b,u6	00101 bbb 11001011FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

ASRSR

Function

Perform an arithmetic shift right operation on an operand. The shift amount is specified in the second operand. Store the rounded and saturated result in the destination register.

Extension Group

DSP basic arithmetic

Operation

$$a = c \gg 0 ? \text{RND32}(b \gg c) : \text{SAT32}(b \ll -c) ;$$

Instruction Format

op a, b, c

Syntax Example

ASRSR<.f> a, b, c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if the result is saturated

Description

Perform an arithmetic shift right operation on an operand. The shift amount is specified in the *c* operand. Store the rounded and saturated result in the destination register. If operand *c* is negative, perform an arithmetic shift left by the shift amount is specified in the *c* operand, and return the saturated result. The left shift saturates if any of the bits shifted out is unequal to the original sign bit or if the result sign bit is different from the original sign bit. The rounding mode is based on the `DSP_CTRL.RM` bits.

The shift amount is saturated to [-32,32].

The saturation flag `DSP_CTRL.SAT` is set if the result of the instruction saturates, regardless of the *.F* suffix. Other flag updates only occur if the set flags suffix (*.F*) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

if (c >= 0) {                                     /* 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	00101bbb00001100FBBBCCCCC
ASRSR<.f>	a,b,u6	00101bbb01001100FBBBuuuuuuAAAAAA
ASRSR<.f>	b,b,s12	00101bbb10001100FBBBssssssSSSSSS
ASRSR<.cc><.f>	b,b,c	00101bbb11001100FBBBCCCCC0QQQQQ
ASRSR<.cc><.f>	b,b,u6	00101bbb11001100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CBFLYHF0R

Function

Compute the first result of a fast Fourier transform (FFT) butterfly operation. Store the rounded and saturated result.

Extension Group

DSP Accumulator

Operation

```

accr = bfly.real + b.r * c.r - b.i * c.i;
acci = bfly.imag + b.i * c.r + b.r * c.i;
a.r  = sat(accr);
a.i  = sat(acci);

```

Instruction Format

op a, b, c

Syntax Example

CBFLYHF0R a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compute the first rounded result of an FFT Butterfly operation, based on the twiddle factor in the accumulator: $A' = A + T * B$.

Store the result in the accumulator. The instruction does set any STATUS32 flags. Optionally, the result can be scaled. The scaling is controlled by the auxiliary register, `DSP_FFT_CTRL`. SC bit. The third butterfly operand is from the auxiliary register, `DSP_BFLY0` register.

This instruction is sensitive to the processor endian mode configuration parameter. In little-endian mode, the low accumulator stores the real part of the result and in the big-endian mode, the low accumulator stores the imaginary part of the result.

An FFT butterfly performs the following operation in which all variables are complex numbers:

```

A1 = A0 + T * B0
B1 = A0 - T * B0 = 2 * A0 - A1

```

The butterfly computations require three inputs and write two outputs. The ARCV2 architecture only supports two explicit input operands to an instruction which is why the A0 input to an FFT butterfly is taken from an auxiliary register (DSP_BFLY0).

```
CBFLYHF0R A1,T,B0      ;compute A1 result using T and B0 explicit input
                        ;operands and DSP_BFLY0 AUX implicit input operand

CBFLYH1FR B1,A0        ; compute B1 result using T A1 and A0 from previous
                        ;iteration and set next A0 value in DSP_BFLY0 AUX
                        ;register.
```

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// Accumulator stores A1 result
if (BYTE_ORDER == little_endian) {
    ACC0_GHI.V = sat((DSP_BFLY0.IMAG<<15) + b.h1 * c.h0 + b.h0 * c.h1, 31,
&ACC0_HI);
    ACC0_GLO.V = sat((DSP_BFLY0.REAL<<15) + b.h0 * c.h0 - b.h1 * c.h1, 31,
&ACC0_LO);
} else {
    ACC0_GHI.V = sat((DSP_BFLY0.REAL<<15) + b.h1 * c.h1 - b.h0 * c.h0, 31,
&ACC0_HI);
    ACC0_GLO.V = sat((DSP_BFLY0.IMAG<<15) + b.h1 * c.h0 + b.h0 * c.h1, 31,
&ACC0_LO);
}
if (DSP_FFT_CTRL.SC) {
    s1 = sat(round(ACC0_HI, 16) >> 16, 15, &a.h1);
    sh = sat(round(ACC0_LO, 16) >> 16, 15, &a.h0);
} else {
    s1 = sat(round(ACC0_HI, 15) >> 15, 15, &a.h1);
    sh = sat(round(ACC0_LO, 15) >> 15, 15, &a.h0);
}
ACC0_GHI.Z = ACC0_HI == 0;
ACC0_GHI.N = ACC0_HI < 0;
ACC0_GHI.G = 0;
ACC0_GLO.Z = ACC0_LO == 0;
ACC0_GLO.N = ACC0_LO < 0;
ACC0_GLO.G = 0;
DSP_CTRL.SAT |= ACC0_GHI.V || ACC0_GLO.V || s1 || sh;
```

Assembly Code Example

```
LD.AB %r5, [%r0, 4]      ; load A
CBFLYHF1R 0, %r5         ; set A into accumulator, ignore return value
LP lpend
LD.AB %r6, [%r1, 4]      ; load B
LD.AB %r6, [%r1, 4]      ; load twiddle factor
LD.AB %r7, [%r2, 4]      ; load B
CBLFYHF0R %r8, %r6, %r7  ; compute A1 = A0 + T * B0
LD.AB %r5, [%r0, 4]      ; next load A
CBFLYHF1R %r9, %r5       ; compute B1 = A0 - T * B0, set next A0 in register
ST.AB %r8, [%r3, 4]      ; write output A1
ST.AB %r9, [%r4, 4]      ; write output B1
lpend:
```

C Code Example

The butterfly computations require three inputs and write two outputs. The ARCV2 ISA supports only two explicit input operands to an instruction. Therefore the A0 input to an FFT butterfly is taken from an auxiliary register (DSP_BFLY0); the A0 value is updated by software pipelining the butterfly computation.

The following example illustrates how these two instructions can be used

```
cbflyhf1r(*ain++);        // first A input, ignore first
for (i = 0; i < max; i++)  return value
{
    *aout++ = cbflyhf0r(*bin++, *twiddle++); // B & twiddle input, A output
    *bout++ = cbflyhf1r(*ain++);           // B output, next A input
}
```

Syntax and Encoding

Instruction Code		
CBFLYHF0R	a, b, c	00110bbb000110111BBBCCCCCAAAAAA
CBFLYHF0R	a, b, u6	00110bbb010110111BBBuuuuuuAAAAAA
CBFLYHF0R	b, b, s12	00110bbb100110111BBBssssssSSSSSS
CBFLYHF0R<.cc>	b, b, c	00110bbb110110111BBBCCCCC0QQQQQ
CBFLYHF0R<.cc>	b, b, u6	00110bbb110110111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

`SUB.F 0, R0, R1`

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CBFLYHF1R

Function

Compute the second result of a fast Fourier transform butterfly operation. Store the rounded and saturated result.

Extension Group

DSP Accumulator

Operation

```
accr = 2 * bfly.real - accr;
acci = 2 * bfly.imag - acci;
b.i = sat(acci); b.r = sat(accr);
bfly = c
```

Instruction Format

op b, c

Syntax Example

CBFLYHF1R b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compute the rounded second result of an FFT Butterfly operation, based on the twiddle factor in the accumulator: $B1 = A0 - T * B = 2 * A0 - A1$.

Store the next A0 in the accumulator. The instruction does set any STATUS32 flags. Optionally, the result can be scaled. The scaling is controlled by the auxiliary register, `DSP_FFT_CTRL.SC` bit. The third butterfly operand is from the auxiliary register, `DSP_BFLY0` register.

This instruction is sensitive to the processor endian mode configuration parameter. In the little-endian mode, the low accumulator stores the real part of the result and in the big-endian mode, the low accumulator stores the imaginary part of the result.

An FFT butterfly performs the following operation in which all variables are complex numbers:

$$A1 = A0 + T * B0$$

$$B1 = A0 - T * B0 = 2 * A0 - A1$$

The butterfly computations require three inputs and write two outputs. The ARCV2 architecture only supports two explicit input operands to an instruction which is why the A0 input to an FFT butterfly is taken from an auxiliary register (DSP_BFLY0).

```
CBFLYHFOR 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                                     /* CBFLYHFR1R*/
if (BYTE_ORDER == little_endian) {
    ACC0_GHI.V = sat((DSP_BFLY0.IMAG << 16) - ACC0_HI, 31, &ACC0_HI);
    ACC0_GLO.V = sat((DSP_BFLY0.REAL << 16) - ACC0_LO, 31, &ACC0_LO);
} else {
    ACC0_GHI.V = sat((DSP_BFLY0.REAL << 16) - ACC0_HI, 31, &ACC0_HI);
    ACC0_GLO.V = sat((DSP_BFLY0.IMAG << 16) - ACC0_LO, 31, &ACC0_LO);
}
if (DSP_FFT_CTRL.SC) {
    sh = sat(round(ACC0_HI, 16) >> 16, 15, &b.h1);
    sl = sat(round(ACC0_LO, 16) >> 16, 15, &b.h0);
} else {
    sh = sat(round(ACC0_HI, 15) >> 15, 15, &b.h1);
    sl = sat(round(ACC0_LO, 15) >> 15, 15, &b.h0);
}
ACC0_GHI.Z = ACC0_HI == 0;
ACC0_GHI.N = ACC0_HI < 0;
ACC0_GHI.G = 0;
ACC0_GLO.Z = ACC0_LO == 0;
ACC0_GLO.N = ACC0_LO < 0;
ACC0_GLO.G = 0;
// set next A0
DSP_BFLY = c;
DSP_CTRL.SAT |= ACC0_GHI.V || ACC0_GLO.V || sl || sh;
```

Assembly Code Example

```
LD.AB %r5, [%r0, 4]      ; load A
CBFLYHF1R 0, %r5         ; set A into accumulator, ignore return value
LP lpend
LD.AB %r6, [%r1, 4]      ; load B
LD.AB %r6, [%r1, 4]      ; load twiddle factor
LD.AB %r7, [%r2, 4]      ; load B
CBFLYHF0R %r8, %r6, %r7  ; compute A1 = A0 + T * B0
LD.AB %r5, [%r0, 4]      ; next load A
CBFLYHF1R %r9, %r5       ; compute B1 = A0 - T * B0, set next A0 in register
ST.AB %r8, [%r3, 4]      ; write output A1
ST.AB %r9, [%r4, 4]      ; write output B1
lpend:
```

C Code Example

The butterfly computations require three inputs and write two outputs. The ARCV2 ISA supports only two explicit input operands to an instruction. Therefore the A0 input to an FFT butterfly is taken from an auxiliary register (DSP_BFLY0); the A0 value is updated by software pipelining the butterfly computation.

The following example illustrates how these two instructions can be used

```
cbflyhf1r(*ain++);          // first A input, ignore first
for (i = 0; i < max; i++)   return value
{
    *aout++ = cbflyhf0r(*bin++,          // B & twiddle input, A output
    *twiddle++);             // B output, next A input
    *bout++ = cbflyhf1r(*ain++);
}
```

Syntax and Encoding

Instruction Code

CBFLYHF1R	b, c	00110 bbb 00 101111 0 BBB CCCCC 011001
CBFLYHF1R	b, limm	00110 bbb 00 101111 0 BBB 111110 011001
CBFLYHF1R	0, c	00110 110 00 101111 0 111 CCCCC 011001
CBFLYHF1R	0, limm	00110 110 00 101111 0 111 111110 011001
CBFLYHF1R	b, u6	00110 bbb 01 101111 0 BBB uuuuuu 011001
CBFLYHF1R	0, u6	00110 110 01 101111 0 111 uuuuuu 011001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMACCHFR

Function

Multiply and accumulate two signed fractional complex conjugate operands. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16(RND16((acci + b.i*c.r - b.r*c.i)<<1));
a.r = SAT16(RND16((accr + b.r*c.r + b.i*c.i)<<1));
```

Instruction Format

op a,b,c

Syntax Example

CMACCHFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply and accumulate the two signed fractional complex conjugate operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        if (BYTE_ORDER == little_endian) {
            addr0 = b.h0 * c.h0 << 1;
            addr1 = b.h1 * c.h1 << 1;
            addi0 = b.h1 * c.h0 << 1;
            addi1 = b.h0 * -c.h1 << 1;
        } else {
            addr0 = b.h1 * -c.h0 << 1;
            addr1 = b.h0 * c.h1 << 1;
            addi0 = b.h0 * c.h0 << 1;
            addi1 = b.h1 * c.h1 << 1;
        }
        acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &alo);
        acchi           = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &ahi);
    else
        // guard bits disabled
        if (BYTE_ORDER == little_endian) {
            s |= sat((b.h0 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h1 * c.h1 << 1), 31, &addr1);
            s |= sat((b.h1 * c.h0 << 1), 31, &addi0);
            s |= sat((b.h0 * -c.h1 << 1), 31, &addi1);
            addi1 = -addi1;
        } else {
            s |= sat((b.h1 * -c.h0 << 1), 31, &addr0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addr1);
            s |= sat((b.h0 * c.h0 << 1), 31, &addi0);
            s |= sat((b.h1 * c.h1 << 1), 31, &addi1);
            addr0 = -addr0;
        }
        acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo, 16), 31, &alo);

```

```

acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    if (BYTE_ORDER == little_endian) {
        addr0 = b.h0 * c.h0;
        addr1 = b.h1 * c.h1;
        addi0 = b.h1 * c.h0;
        addi1 = b.h0 * -c.h1;
    } else {
        addr0 = b.h1 * -c.h0;
        addr1 = b.h0 * c.h1;
        addi0 = b.h0 * c.h0;
        addi1 = b.h1 * c.h1;
    }
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1,16), 31, &alo);
        acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMACCHFR r0,r1,r2    ;Signed complex conjugated multiply-accumulate
                     ; two complex vectors r1 and r2 and
                     ; return the saturated and rounded result in r0
```

Syntax and Encoding

Instruction Code		
CMACCHFR	a, b, c	00110bbb000010011BBBCCCCC0AAAAA
CMACCHFR	a, b, u6	00110bbb010010011BBBuuuuuu0AAAAA
CMACCHFR	b, b, s12	00110bbb100010011BBBsssssssSSSSSS
CMACCHFR<.cc>	b, b, c	00110bbb110010011BBBCCCCC0QQQQQ
CMACCHFR<.cc>	b, b, u6	00110bbb110010011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMACCHNFR

Function

Multiply and accumulate two signed fractional complex conjugate operands. The result is rounded and saturated but is returned without a fractional shift.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16((acci + b.i*c.r - b.r*c.i));
a.r = SAT16((accr + b.r*c.r + b.i*c.i));
```

Instruction Format

op a,b,c

Syntax Example

CMACCHNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply and accumulate two signed fractional complex conjugate operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector without a fractional shift in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0;
    addr1 = b.h1 * c.h1;
    addi0 = b.h1 * c.h0;
    addi1 = b.h0 * -c.h1;
} else {
    addr0 = b.h1 * -c.h0;
    addr1 = b.h0 * c.h1;
    addi0 = b.h0 * c.h0;
    addi1 = b.h1 * c.h1;
}
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
    &ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
    &ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
    &ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
    &ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);
```

Assembly Code Example

```
CMACCHNFR r0,r1,r2    ; Signed complex multiply-accumulate
                      ; two complex vectors r1 and r2 and
                      ; return result in r0 with saturation and rounding
                      ; and returned without a fractional shift
```

Syntax and Encoding

		Instruction Code
CMACCHNFR	a, b, c	00110bbb000010001BBBCCCCCAAAAAA
CMACCHNFR	a, b, u6	00110bbb010010001BBBuuuuuuAAAAAA
CMACCHNFR	b, b, s12	00110bbb100010001BBBssssssSSSSSS
CMACCHNFR<.cc>	b, b, c	00110bbb110010001BBBCCCCC0QQQQQ
CMACCHNFR<.cc>	b, b, u6	00110bbb110010001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMACHFR

Function

Multiply and accumulate two signed fractional complex operands. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16(RND16((acci + b.i*c.r + b.r*c.i)<<1));
a.r = SAT16(RND16((accr + b.r*c.r - b.i*c.i)<<1));
```

Instruction Format

op a,b,c

Syntax Example

CMACHFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply and accumulate the two signed fractional complex operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        if (BYTE_ORDER == little_endian) {
            addr0 = b.h0 * c.h0 << 1;
            addr1 = b.h1 * -c.h1 << 1;
            addi0 = b.h1 * c.h0 << 1;
            addi1 = b.h0 * c.h1 << 1;
        } else {
            addr0 = b.h1 * c.h0 << 1;
            addr1 = b.h0 * c.h1 << 1;
            addi0 = b.h0 * -c.h0 << 1;
            addi1 = b.h1 * c.h1 << 1;
        }
        acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z,
&ACC0_GLO.N, &ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        s                 |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        s                 |= sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        if (BYTE_ORDER == little_endian) {
            s |= sat((b.h0 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h1 * -c.h1 << 1), 31, &addr1);
            s |= sat((b.h1 * c.h0 << 1), 31, &addi0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addi1);
            addr1 = -addr1;
        } else {
            s |= sat((b.h1 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addr1);
            s |= sat((b.h0 * -c.h0 << 1), 31, &addi0);
            s |= sat((b.h1 * c.h1 << 1), 31, &addi1);
            addi0 = -addi0;
        }
        acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z,
&ACC0_GLO.N, &ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        s                 |= sat(round(acclo, 16), 31, &alo);

```

```

acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
  ACC0_GHI.G      = 0;
  ACC0_HI         = acchi & 0xffff_ffff;
  s               |= sat(round(acchi, 16), 31, &ahi);
else
  // post-accumulate fractional shift mode
  if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0;
    addr1 = b.h1 * -c.h1;
    addi0 = b.h1 * c.h0;
    addi1 = b.h0 * c.h1;
  } else {
    addr0 = b.h1 * c.h0;
    addr1 = b.h0 * c.h1;
    addi0 = b.h0 * -c.h0;
    addi1 = b.h1 * c.h1;
  }
  if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z,
&ACC0_GLO.N, &ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi<<1, 16), 31, &ahi);
  else
    // guard bits are disabled
    acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z,
&ACC0_GLO.N, &ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi<<1, 16), 31, &ahi);
  DSP_CTRL.SAT |= s | ACC0_GLO.V | ACC0_GHI.V;
  a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMACHFR r0,r1,r2      ;Signed complex conjugated multiply-accumulate
                       ; two complex vectors r1 and r2 and
                       ; return the saturated and rounded result in r0
```

Syntax and Encoding

Instruction Code

CMACHFR	a, b, c	00110bbb000001111BBBCCCCCAAAAAA
CMACHFR	a, b, u6	00110bbb010001111BBBuuuuuuAAAAAA
CMACHFR	b, b, s12	00110bbb100001111BBBssssssSSSSSS
CMACHFR<.cc>	b, b, c	00110bbb110001111BBBCCCCC0QQQQQ
CMACHFR<.cc>	b, b, u6	00110bbb110001111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMACHNFR

Function

Multiply and accumulate two signed fractional complex operands. The result is rounded and saturated, and is returned without a fractional shift.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16((acci + b.i*c.r + b.r*c.i));
a.r = SAT16((accr + b.r*c.r - b.i*c.i));
```

Instruction Format

op a,b,c

Syntax Example

CMACHNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply and accumulate two signed fractional complex operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector without any fractional shift in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0;
    addr1 = b.h1 * -c.h1;
    addi0 = b.h1 * c.h0;
    addi1 = b.h0 * c.h1;
} else {
    addr0 = b.h1 * c.h0;
    addr1 = b.h0 * c.h1;
    addi0 = b.h0 * -c.h0;
    addi1 = b.h1 * c.h1;
}
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V ;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);
```

CMACHNFR

Assembly Code Example

```
CMACHNFR r0,r1,r2    ;Signed complex multiply-accumulate
                     ; two complex vectors r1 and r2 and
                     ; return result in r0 with saturation and rounding
                     ;without any fractional shift
```

Syntax and Encoding

Instruction Code

CMACHNFR	a, b, c	00110bbb000001101BBBCCCCCAAAAAA
CMACHNFR	a, b, u6	00110bbb010001101BBBuuuuuuAAAAAA
CMACHNFR	b, b, s12	00110bbb100001101BBBssssssSSSSSS
CMACHNFR<.cc>	b, b, c	00110bbb110001101BBBCCCCC0QQQQQ
CMACHNFR<.cc>	b, b, u6	00110bbb110001101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYCHFR

Function

Multiply two signed fractional complex conjugate operands. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.h1 = SAT16(RND16((b.i*c.r - b.r*c.i)<<1));
a.h0 = SAT16(RND16((b.r*c.r + b.i*c.i)<<1));
```

Instruction Format

op a,b,c

Syntax Example

CMPYCHFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply two signed fractional complex conjugate operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
if (DSP_CTRL.PA)
// pre-accumulate fractional shift mode
if (DSP_CTRL.GE)
// guard bits are enabled
if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0 << 1;
    addr1 = b.h1 * c.h1 << 1;
    addi0 = b.h1 * c.h0 << 1;
    addi1 = b.h0 * -c.h1 << 1;
} else {
    addr0 = b.h1 * -c.h0 << 1;
    addr1 = b.h0 * c.h1 << 1;
    addi0 = b.h0 * c.h0 << 1;
    addi1 = b.h1 * c.h1 << 1;
}
acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
ACC0_LO        = acclo & 0xffff_ffff;
s              |= sat(round(acclo, 16), 31, &alo);
acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
ACC0_HI        = acchi & 0xffff_ffff;
S              |= sat(round(acclo, 16), 31, &ahi);
else
// guard bits disabled
if (BYTE_ORDER == little_endian) {
    s |= sat((b.h0 * c.h0 << 1), 31, &addr0);
    s |= sat((b.h1 * c.h1 << 1), 31, &addr1);
    s |= sat((b.h1 * c.h0 << 1), 31, &addi0);
    s |= sat((b.h0 * -c.h1 << 1), 31, &addi1);
    addi1 = -addi1;
} else {
    s |= sat((b.h1 * -c.h0 << 1), 31, &addr0);
    s |= sat((b.h0 * c.h1 << 1), 31, &addr1);
    s |= sat((b.h0 * c.h0 << 1), 31, &addi0);
    s |= sat((b.h1 * c.h1 << 1), 31, &addi1);
    addr0 = -addr0;
}
acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
ACC0_GLO.G      = 0;
ACC0_LO        = acclo & 0xffff_ffff;
s              |= sat(round(acclo, 16), 31, &alo);

```

```

    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    if (BYTE_ORDER == little_endian) {
        addr0 = b.h0 * c.h0;
        addr1 = b.h1 * c.h1;
        addi0 = b.h1 * c.h0;
        addi1 = b.h0 * -c.h1;
    } else {
        addr0 = b.h1 * -c.h0;
        addr1 = b.h0 * c.h1;
        addi0 = b.h0 * c.h0;
        addi1 = b.h1 * c.h1;
    }
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1,16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 00
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);
    DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V;
    a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMPYCHFR r0,r1,r2    ;Signed complex conjugated multiply
                    ; two complex vectors r1 and r2 and
                    ; return result in r0 with saturation and rounding
```

Syntax and Encoding

Instruction Code		
CMPYCHFR	a, b, c	00110bbb000001011BBBCCCCCAAAAAA
CMPYCHFR	a, b, u6	00110bbb010001011BBBuuuuuuAAAAAA
CMPYCHFR	b, b, s12	00110bbb100001011BBBssssssSSSSSS
CMPYCHFR<.cc>	b, b, c	00110bbb110001011BBBCCCCC0QQQQQ
CMPYCHFR<.cc>	b, b, u6	00110bbb110001011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYCHNFR

Function

Multiply two signed fractional complex conjugate operands. The result is rounded and saturated and returned without a fractional shift.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16((b.i*c.r - b.r*c.i));
a.r = SAT16(( b.i*c.i + b.i*c.i));
```

Instruction Format

op a,b,c

Syntax Example

CMPYCHNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply two signed fractional complex conjugate operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator. Return the saturated and rounded complex vector without a fractional shift in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
= 0;
addr0 = b.h0 * c.h0;
addr1 = b.h1 * c.h1;
addi0 = b.h1 * c.h0;
addi1 = b.h0 * -c.h1;
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```

CMPYCHNFR r0,r1,r2    ;Signed complex conjugated multiply
                      ; two complex vectors r1 and r2 and
                      ; return result in r0 with saturation and rounding
                      ;and returned without a fractional shifting

```

Syntax and Encoding

		Instruction Code
CMPYCHNFR	a, b, c	00110bbb000000101BBBCCCCCAAAAAA
CMPYCHNFR	a, b, u6	00110bbb010000101BBBuuuuuuAAAAAA
CMPYCHNFR	b, b, s12	00110bbb100000101BBBssssssSSSSSS
CMPYCHNFR<.cc>	b, b, c	00110bbb110000101BBBCCCCC0QQQQQ
CMPYCHNFR<.cc>	b, b, u6	00110bbb110000101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYHFR

Function

Multiply two signed fractional complex operands. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16 (RND16 ((b.i*c.r + b.r*c.i)<<1));
a.r = SAT16 (RND16 ((b.r*c.r - b.i*c.i)<<1));
```

Instruction Format

op a,b,c

Syntax Example

CMPYHFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply two signed fractional complex operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in the little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in the big-endian mode. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        if (BYTE_ORDER == little_endian) {
            addr0 = b.h0 * c.h0 << 1;
            addr1 = b.h1 * -c.h1 << 1;
            addi0 = b.h1 * c.h0 << 1;
            addi1 = b.h0 * c.h1 << 1;
        } else {
            addr0 = b.h1 * c.h0 << 1;
            addr1 = b.h0 * c.h1 << 1;
            addi0 = b.h0 * -c.h0 << 1;
            addi1 = b.h1 * c.h1 << 1;
        }
        acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        s                 |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        s                 |= sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        if (BYTE_ORDER == little_endian) {
            s |= sat((b.h0 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h1 * -c.h1 << 1), 31, &addr1);
            s |= sat((b.h1 * c.h0 << 1), 31, &addi0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addi1);
            addr1 = -addr1;
        } else {
            s |= sat((b.h1 * c.h0 << 1), 31, &addr0);
            s |= sat((b.h0 * c.h1 << 1), 31, &addr1);
            s |= sat((b.h0 * -c.h0 << 1), 31, &addi0);
            s |= sat((b.h1 * c.h1 << 1), 31, &addi1);
            addi0 = -addi0;
        }

```

```

    acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    if (BYTE_ORDER == little_endian) {
        addr0 = b.h0 * c.h0;
        addr1 = b.h1 * -c.h1;
        addi0 = b.h1 * c.h0;
        addi1 = b.h0 * c.h1;
    } else {
        addr0 = b.h1 * c.h0;
        addr1 = b.h0 * c.h1;
        addi0 = b.h0 * -c.h0;
        addi1 = b.h1 * c.h1;
    }
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s               |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s               |= sat(round(acchi<<1, 16), 31, &ahi);

```

```

else
    // guard bits are disabled
    acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;0
    ACC0_LO         = acclo & 0xffff_ffff;
    s               |= sat(round(acclo<<1,16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s               |= sat(round(acchi<<1, 16), 31, &ahi);
    DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V ;
    a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```

CMPYHFR r0,r1,r2      ;Signed complex multiply
                      ; two complex vectors r1 and r2 and
                      ; return result in r0 with saturation and rounding

```

Syntax and Encoding

Instruction Code		
CMPYHFR	a, b, c	00110bbb00000011BBBCCCCCAAAAAA
CMPYHFR	a, b, u6	00110bbb010000011BBBuuuuuuAAAAAA
CMPYHFR	b, b, s12	00110bbb100000011BBBssssssSSSSSS
CMPYHFR<.cc>	b, b, c	00110bbb110000011BBBCCCCC0QQQQQ
CMPYHFR<.cc>	b, b, u6	00110bbb110000011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYHFMR

Function

Scale a signed fractional complex operand. The result is rounded and saturated.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16 (b.i*c.i) <<1 ;
a.r = SAT16 ((b.r*c.i) <<1) ;
```

Instruction Format

op a,b,c

Syntax Example

CMPYHFMR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Scale a signed fractional complex operand, *b*, with the scalar fractional value specified in the *c* operand. Store the real result in the low accumulator and store the imaginary result in the high accumulator. Return the saturated and rounded complex vector in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addr = b.h0 * c.h1 << 1;
        addi = b.h1 * c.h1 << 1;
        acclo = acc_add(0, addr, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s                |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s                |= sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h1 << 1), 31, &addr);
        s |= sat((b.h1 * c.h1 << 1), 31, &addi);
        acclo = acc_add(0, addr, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s                |= sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s                |= sat(round(acchi, 16), 31, &ahi);

```

CMPYHFMR

```

else
    // post-accumulate fractional shift mode
    addr = b.h0 * c.h1;
    addi = b.h1 * c.h1;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acclo = acc_add(0, addr, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s                |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(0, addi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s                |= sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acclo = acc_add(0, addr, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO         = acclo & 0xffff_ffff;
        s                |= sat(round(acclo<<1, 16), 31, &alo);
        acchi = acc_add(0, addi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI         = acchi & 0xffff_ffff;
        s                |= sat(round(acchi<<1, 16), 31, &ahi);
    DSP_CTRL.SAT |= s | ACC0_GHI.V | ACC0_GLO.V;
    a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```

CMPYHFMR r0,r1,r2    ;Signed complex scaling of r1 with scalar r2
                    ; return result in r0 with saturation and rounding

```

Syntax and Encoding

Instruction Code		
CMPYHFMR	a,b,c	00110 bbb 00 011011 0BBBCCCCC AAAAAA
CMPYHFMR	a,b,u6	00110 bbb 01 011011 0BBBuuuuuu AAAAAA
CMPYHFMR	b,b,s12	00110 bbb 10 011011 0BBBssssss SSSSSS

CMPYHFMR<.cc>	b, b, c	00110bbb110110110BBBCCCCC0QQQQQ
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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

CMPYHNFR

Function

Multiply two signed fractional complex operands. The result is rounded and saturated, and returned without a fractional shift.

Extension Group

DSP Complex MAC

Operation

```
a.i = SAT16((b.i*c.r + b.r*c.i));
a.r = SAT16((b.r*c.r - b.i*c.i));
```

Instruction Format

op a,b,c

Syntax Example

CMPYHNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Multiply two signed fractional complex operands. Store the real result in the low accumulator and store the imaginary result in the high accumulator in a little-endian mode. Store the real result in the high accumulator and store the imaginary result in the low accumulator in a big-endian mode. Return the saturated and rounded complex vector without fractional shift in the destination operand. This instruction sets the sticky flag, `DSP_CTRL.SAT`, if any of the real or imaginary components saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (BYTE_ORDER == little_endian) {
    addr0 = b.h0 * c.h0;
    addr1 = b.h1 * -c.h1;
    addi0 = b.h1 * c.h0;
    addi1 = b.h0 * c.h1;
} else {
    addr0 = b.h1 * c.h0;
    addr1 = b.h0 * c.h1;
    addi0 = b.h0 * -c.h0;
    addi1 = b.h1 * c.h1;
}
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addr0+addr1, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addr0+addr1, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s                |= sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(0, addi0+addi1, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    s                |= sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT = s | ACC0_GHI.V | ACC0_GLO.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) >> 0x0000_ffff);

```

Assembly Code Example

```
CMPLYHNFR r0,r1,r2    ;Signed complex multiply
                      ; two complex vectors r1 and r2 and
                      ; return result in r0 with saturation and rounding
                      ;and returned without a fractional shift
```

Syntax and Encoding

		Instruction Code
CMPLYHNFR	a, b, c	00110bbb00000001BBBCCCCCAAAAAA
CMPLYHNFR	a, b, u6	00110bbb010000001BBBuuuuuuAAAAAA
CMPLYHNFR	b, b, s12	00110bbb100000001BBBssssssSSSSSS
CMPLYHNFR<.cc>	b, b, c	00110bbb110000001BBBCCCCC0QQQQQ
CMPLYHNFR<.cc>	b, b, u6	00110bbb110000001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DIV

Function

2's Complement Integer Divide. For the non-DSP version, see [DIV](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

if (cc) then $a = b / c$;

Instruction Format

op a, b, c

Syntax Example

DIV <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if divisor is zero or if an arithmetic overflow occurs

Description

If the divisor (c) is non-zero, the destination register is assigned the quotient obtained by signed integer division of source operand (b) by source operand (c).

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

An arithmetic overflow is signaled if the resulting value cannot be represented in 32 bits. This overflow occurs only in one specific case, which is the division of the largest representable negative value (0x80000000) by -1. If an arithmetic overflow occurs, the overflow flag (V) is set to 1.

If a division-by-zero is attempted, or if the result cannot be represented in 32 bits, when the EV_DivZero exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, “Implementation Behavior on Division Overflow”](#).

Pseudo Code

```

if (cc == true) {
    if ((src2 != 0) && ((src1 != 0x80000000) || src2 != 0xffffffff))
    {
        dest = src1 / src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = dest[31];
            V = 0;
        }
    } else {
        if ((src2 == 0) && (STATUS32.DZ == 1))
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1) {
                V = 1;
            }
        }
    }
}

```

Assembly Code Example

```
DIV r1,r2,r3 ; r1 is assigned r2 / r3
```

Syntax and Encoding

Instruction Code

DIV<.f>	a,b,c	00101bbb00000100FBBBCCCCCAAAAAA
DIV<.f>	a,b,u6	00101bbb01000100FBBBuuuuuuAAAAAA
DIV<.cc><.f>	b,b,c	00101bbb11000100FBBBCCCCC0QQQQQ
DIV<.cc><.f>	b,b,u6	00101bbb11000100FBBBuuuuuu1QQQQQ
DIV<.f>	b,b,s12	00101bbb10000100FBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DIVF

Function

Signed 32-bit fractional division.

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

See the pseudo code for information about the operation.

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 division of the 32-bits operands b and c, assuming $\text{abs}(b) \leq \text{abs}(c)$ or $B == -C$, returning the quotient to a.

The EV_DivZero exception is raised if the divisor is 0 and the STATUS32.DZ bit is set. If the absolute value of the numerator is greater than the absolute value of the denominator, or if the denominator and numerator are equal, the result returned is 0x80000000 if the operand signs differ or 0x7fffffff if the operand signs are equal. If the divisor is 0, the destination register is not updated.

Status flags are updated only if the .F suffix is specified, in which case: if the divisor is 0 the Z, and N flags are cleared and the V flag is set; otherwise Z is set if the quotient is zero, N is set if the quotient is negative, and V is cleared.

Pseudo Code

```
if (c != 0) /*DIVF*/
{
    if ((abs(b) < abs(c)) || (b == -c))
    {
        a = (b<<31) / c;
    }
    else
    {
        a = (b < 0) == (c < 0) ? 0x7fffffff :
0x80000000;
    }
    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	00110bbb00010000FBBBCCCCCAAAAAA
DIVF<.f>	a,b,u6	00110bbb01010000FBBBuuuuuuAAAAAA
DIVF<.f>	b,b,s12	00110bbb10010000FBBBssssssSSSSSS

DIVF<cc><.f>	b, b, u6	00110 bbb 11010000 FBBB uuuuuu 1 QQQQQ
DIVF<.cc><.f>	b, b, c	00110 110 11010000 F111 111110 0 QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DIVU

Function

Unsigned integer division. For the non-DSP version, see [DIVU](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

if (cc) then $a = b / c$;

Instruction Format

op a, b, c

Syntax Example

DIVU <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Always cleared
C		= Unchanged
V	•	= Set if divisor is zero

Description

If the divisor (c) is non-zero, the destination register is assigned the quotient obtained by unsigned integer division of source operand (b) by source operand (c).

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

If a division- by-zero is attempted, when the EV_DivZero exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, “Implementation Behavior on Division Overflow”](#).

If the flag-update bit is set, an unsigned DIVU operation always clears the N-flag.

Pseudo Code

```

if (cc == true) {
    if (src2 != 0) {
        dest = src1 / src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = 0;
            V = 0;
        }
    } else {
        if (STATUS32.DZ == 1)
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1) {
                V = 1;
            }
        }
    }
}
/* DIVU */

```

Assembly Code Example

```
DIVU r1,r2,r3 ; r1 is assigned r2 / r3
```

Syntax and Encoding

Instruction Code

DIVU<.f>	a,b,c	00101 bbb 00 000101 F BBB CCCCC AAAAAA
DIVU<.f>	a,b,u6	00101 bbb 01 000101 F BBB uuuuuu AAAAAA
DIVU<.cc><.f>	b,b,c	00101 bbb 11 000101 F BBB CCCCC 0QQQQQ
DIVU<.cc><.f>	b,b,u6	00101 bbb 11 000101 F BBB uuuuuu 1QQQQQ
DIVU<.f>	b,b,s12	00101 bbb 10 000101 F BBB ssssss SSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACH

Function

Dual 16x16 signed integer multiply and accumulate. For the non-DSP version, see [DMACH](#).

Extension Group

MPY_OPTION == 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
    a = result.w0;
    acc = result;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. w0 represents the lower 32-bits of an operand.

Instruction Format

op a, b, c

Syntax Example

DMACH <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	= Set if an overflow occurs during accumulation. This instruction never clears this flag.

Description

Multiply the lower 16 bits of the first and second operands, and multiply the higher 16 bits of the first and second operands to generate two 32-bit products. The two products are added to the wide accumulator to form the result. The least-significant 32 bits of the wide accumulator are assigned to the destination register. Flags are updated only if the set flags suffix (.F) is used.

Pseudo Code

```
accwide = accwide_org + b.h0 * c.h0 + b.h1 * c.h1;           /* DMACH */
if (DSP_CTRL.GE) // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    V_flag |= rem != 0 && rem != -1;
    N_flag = accwide < 0;
}
```

Assembly Code Example

```
DMACH r1,r2,r3 ; dual 16x16 MAC of r2 and r3
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

Instruction Code		
DMACH<.f>	a,b,c	00101bbb00010010FBBBCCCCCAAAAAA
DMACH<.f>	a,b,u6	00101bbb01010010FBBBuuuuuuAAAAAA
DMACH<.f>	b,b,s12	00101bbb10010010FBBBssssssSSSSSS
DMACH<.cc><.f>	b,b,c	00101bbb11010010FBBBCCCCC0QQQQQ
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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHBL

Function

Dual 16x8 signed integer multiply-accumulate operation of 16-bit vector elements of the first operand and the sign-extended first and second bytes of the second operand.

Extension Group

DSP Dual 16x8 MAC

Operation

$$a.h1 = accwide + b.h1 * sext(c.b1) + b.h0 * sext(c.b0);$$

Instruction Format

op a, b, c

Syntax Example

DMACHFBL <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	=Set if the accumulator overflows; this instruction never clears this flag

Description

Perform a signed integer multiplication of the higher 16 bits of the *b* operand and the sign-extended second byte of the *c* operand. Perform a signed integer multiplication the lower 16 bits of the *b* operand with the sign-extended first byte of the *c* operand. Add the two products to the wide accumulator and store the result in the wide accumulator. Return the least-significant 32 bits of the accumulator as the result. Flags are only updated if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

accwide += b.h0 * c.b0 + b.h1 * c.b1;                                /* DMACHBL
if (DSP_CTRL.GE) // only if guard bits are enabled                  */
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    V_flag |= rem != 0 && rem != -1;
    N_flag = accwide < 0;
}

```

Assembly Code Example

```

DMACHBL r0,r2,r3           ; Signed integer multiply-accumulate two
                           ;vectors r2 and r3 add the products ;into the
                           ;wide accumulator and return ;result in r0

```

Syntax and Encoding

Instruction Code

DMACHBL<.f>	a, b, c	00110 bbb 00 011000 FBBBCCCCC AAAAAA
DMACHBL<.f>	a, b, u6	00110 bbb 01 011000 FBBBuuuuuu AAAAAA
DMACHBL<.f>	b, b, s12	00110 bbb 10 011000 FBBBssssss SSSSSS
DMACHBL<.cc><.f>	b, b, c	00110 bbb 11 011000 FBBBCCCCC 0QQQQQ
DMACHBL<.cc><.f>	b, b, u6	00110 bbb 11 011000 FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHBM

Function

Dual 16x8 signed integer multiply-accumulate operation of 16-bit vector elements of the first operand and the sign-extended third and fourth bytes of the second operand.

Extension Group

DSP Dual 16x8 MAC

Operation

$$a.h1 = accwide + b.h1 * sext(c.b3) + b.h0 * sext(c.b2);$$

Instruction Format

op a, b, c

Syntax Example

DMACHBM <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	=Set if the accumulator overflows; this instruction never clears this flag

Description

Signed integer multiplication of the higher 16 bits of the *b* operand and the sign-extended fourth byte of the *c* operand; signed integer multiplication the lower 16 bits of the *b* operand with the sign-extended third byte of the *c* operand. Add the two products to the wide accumulator and store the result in the wide accumulator. Return the least-significant 32 bits of the accumulator as the result. Flags are only updated if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

accwide += b.h0 * c.b2 + b.h1 * c.b3;                                /* DMACHBM
if (DSP_CTRL.GE) // only if guard bits are enabled                  */
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    V_flag |= rem != 0 && rem != -1;
    N_flag = accwide < 0;
}

```

Assembly Code Example

```

DMACHBM r0,r2,r3           ; Signed integer multiply-accumulate two
                           ;vectors r2 and r3 add the products ;into the
                           ;wide accumulator and return ;result in r0

```

Syntax and Encoding

Instruction Code

DMACHBM<.f>	a, b, c	00110bbb00011001FBBBCCCCC(AAAAAA)
DMACHBM<.f>	a, b, u6	00110bbb01011001FBBBuuuuuu(AAAAAA)
DMACHBM<.f>	b, b, s12	00110bbb10011001FBBBssssss(SSSSSS)
DMACHBM<.cc><.f>	b, b, c	00110bbb11011001FBBBCCCCC(000000)
DMACHBM<.cc><.f>	b, b, u6	00110bbb11011001FBBBuuuuuu(100000)

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

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

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result is too big to be stored in the destination

Description

Multiply the lower 16 bits of the first and second operands, and multiply the higher 16 bits of the first and second operands to generate two 32-bit products. The two products are added to the high accumulator to form the result. Return the 16-bit saturated fractional result to the destination register. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* DMACHF
s = 0;                                                                           */
if (DSP_CTRL.PA)                                                                */
// pre-accumulate fractional shift mode
if (DSP_CTRL.GE)
// guard bits are enabled
addlo = b.h0 * c.h0 << 1;
addhi = b.h1 * c.h1 << 1;
acchi = acc_add(acchi, addlo+addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 : ACC0_GHI.GUARD != 0;
ACC0_HI = acchi & 0xffff_ffff;
v = sat(acchi, 31, &ahi);
else
// guard bits disabled
s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
acchi = acc_add(acchi, addhi+addlo, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.G = 0;
ACC0_HI = acchi & 0xffff_ffff;
v = sat(acchi, 31, &ahi)
else
// post-accumulate fractional shift mode
addlo = b.h0 * c.h0;
addhi = b.h1 * c.h1;
if (DSP_CTRL.GE)
// guard bits are enabled
acchi = acc_add(acchi, addhi+addlo, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 && ACC0_GHI.GUARD !=
0;
ACC0_HI = acchi & 0xffff_ffff;
v = sat(acchi<<1, 31, &ahi);
else
// guard bits are disabled
acchi = acc_add(acchi, addhi+addlo, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
ACC0_GHI.G = 0;
ACC0_HI = acchi & 0xffff_ffff;
v = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_0000);
if F {
N_flag = a < 0;
V_flag = v;
Z_flag = a == 0;
}

```

Assembly Code Example

```

DMACHF r0,r1,r2                                ;signed fractional multiply-
                                                ;accumulate of two 16x16 vectors.
                                                ;The saturated result is stored
                                                ;in r0

```

Syntax and Encoding

		Instruction Code
DMACHF<.f>	a, b, c	00101bbb00101100FBBBCCCCC(AAAAAA)
DMACHF<.f>	a, b, u6	00101bbb01101100FBBBuuuuuu(AAAAAA)
DMACHF<.f>	b, b, s12	00101bbb10101100FBBBssssss(SSSSSS)
DMACHF<.cc><.f>	b, b, c	00101bbb11101100FBBBCCCCC(0QQQQQ)
DMACHF<.cc><.f>	b, b, u6	00101bbb11101100FBBBuuuuuu(1QQQQQ)

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHFR

Function

Dual 16x16 signed fractional multiplication and accumulation. The result is rounded and saturated.

Extension Group

DSP dual 16x16 MAC

Operation

$$a.h1 = \text{SAT16}(\text{RND16}((acchi += (b.h1 * c.h1 + b.h0 * c.h0) << 1)));$$


Note

$h0$ (lower) and $h1$ (upper) represent the 16-bit elements of a 32-bit operand. $w0$ represents the lower 32-bits of an operand.

Instruction Format

op a, b, c

Syntax Example

DMACHFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result is too big to be stored in the destination

Description

Multiply the lower 16 bits of the first and second operands, and multiply the higher 16 bits of the first and second operands. Add the two products to the high accumulator to form the result. Return the 16-bit saturated and rounded fractional result to the destination register. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(acchi, addlo+addhi, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(acchi, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    addlo = b.h0 * c.h0;
    addhi = b.h1 * c.h1;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acchi = acc_add(acchi, addhi+addlo, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acchi = acc_add(acchi, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_0000);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}

```

Assembly Code Example

```
DMACHFR r0,r1,r2           ;signed fractional multiply-
                           ;accumulate of two 16x16 vectors.
                           ;The saturated and rounded result
                           ;is stored ;in r0
```

Syntax and Encoding

Instruction Code

DMACHFR<.f>	a,b,c	00101bbb00101101FBBBCCCCC	AAAAAA
DMACHFR<.f>	a,b,u6	00101bbb01101101FBBBuuuuuu	AAAAAA
DMACHFR<.f>	b,b,s12	00101bbb10101101FBBBssssss	SSSSSS
DMACHFR<.cc><.f>	b,b,c	00101bbb11101101FBBBCCCCC	0QQQQQ
DMACHFR<.cc><.f>	b,b,u6	00101bbb11101101FBBBuuuuuu	100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACHU

Function

Dual unsigned integer 16x16 multiply and accumulate. For the non-DSP version, see [DMACH](#).

Extension Group

MPY_OPTION == 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
    a = result.w0;
    acc = result;
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. `w0` represents the lower 32-bits of an operand.

Instruction Format

op a, b, c

Syntax Example

DMACHU <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Set if an overflow occurs during accumulation. This instruction never clears this flag.

Description

Multiply the lower 16 bits of the first and second operands, and multiply the higher 16 bits of the first and second operands. The two products are added to the wide accumulator to form the result. The least-significant 32 bits of the wide accumulator are assigned to the destination register. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
accwide = accwide_org + b.h0 * c.h0 + b.h1 * c.h1;           /* DMACHU */
if (DSP_CTRL.GE)      // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 72
else
    rem = accwide >> 64
if F {
    V_flag |= rem != 0
}
```

Assembly Code Example

```
DMACHU r1,r2,r3                ; dual 16x16 unsigned MAC of r2
                                ;and r3
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
DMACHU<.f>	a,b,c	00101bbb00010011FBBBCCCCCAAAAAA
DMACHU<.f>	a,b,u6	00101bbb01010011FBBBuuuuuuAAAAAA
DMACHU<.f>	b,b,s12	00101bbb10010011FBBBssssssSSSSSS
DMACHU<.cc><.f>	b,b,c	00101bbb11010011FBBBCCCCC0QQQQQ
DMACHU<.cc><.f>	b,b,u6	00101bbb11010011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACWH

Function

Dual 32x16 multiply and accumulate. For the non-DSP version, see [DMACWH](#).

Extension Group

MPY_OPTION == 9 or dsp_wide==true

Operation

```
if (cc) {
    accwide += (b.w0 * c.h0 + b.w1 * c.h1);
    a = accwide;
}
```



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 and B are 64-bit operands, and must be specified as even numbered registers. For more information about 64-bit operands, see ["Data Formats"](#).

Instruction Format

op A, B, c

Syntax Example

DMACWH <.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 sum of the accumulator and the two 32x16 products formed by multiplying B.w0 with c.h0, and B.w1 with c.h1. The resulting sum is assigned to the destination register pair, and is also assigned to the 64-bit accumulator (ACCH, ACCL). In common with all operations that produce a 64-bit result, the 64-bit operands must be an even-numbered register (or the null destination).

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

```
// note: accumulators modeled as 128b integers          /* DMACWH */
if (DSP_CTRL.GE)
    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.w0 * c.ho + b.w1 * c.h1;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = accwide & 0xffff_ffff_ffff_ffff;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff_ffff_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

```
DMACWH r0,r2,r5                                ; dual 32x16 MAC of r2 and r5
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
DMACWH<.f>	a,b,c	00101bbb00110110FBBBCCCCCAAAAAA
DMACWH<.f>	a,b,u6	00101bbb01110110FBBBuuuuuuAAAAAA
DMACWH<.f>	b,b,s12	00101bbb10110110FBBBsssssssSSSSSS
DMACWH<.cc><.f>	b,b,c	00101bbb11110110FBBBCCCCC0QQQQQ
DMACWH<.cc><.f>	b,b,u6	00101bbb11110110FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACWHF

Function

Dual 32x16 fractional multiply and accumulate

Extension Group

dsp_wide==true

Operation

```
if (cc) {
    accwide += (B.w0 * c.h0 + B.w1 * c.h1) <<16;
    a = SAT32(accwide>>31);
}
```



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. B is a 64-bit operand, and must be specified as even numbered registers. For more information about 64-bit operands, see [“Data Formats”](#).

Instruction Format

op a, B, c

Syntax Example

DMACWH <.f> a, B, c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
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 sum of the accumulator and the two 32x16 fractional products formed by multiplying $B.w0$ with $c.h0$, and $B.w1$ with $c.h1$. The resulting sum is assigned to the 64-bit accumulator (ACCH, ACCL). The saturated 32-bit value of the accumulator is returned in the destination register.

The sticky DSP_CTRL.SAT flag is set if saturation occurred, regardless of the .F suffix. The other flags are updated if the .F suffix is used. Flags are updated only if the set flags suffix (.F) is used.

Pseudo Code

```
// note: accumulators modeled as 128b integers          /* DMACWHF */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add0 = b.w0 * c.h0 << 17;
        add1 = b.w1 * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 |
unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add0 + add1,
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.w0 * c.h0 << 17), 63, &add0) |
sat((b.w1 * c.h1 << 17), 63, &add0);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add0 + add1, 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.w0 * c.h0 +
b.w1 * 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.w0 * c.h0 + b.w1 *
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

```
DMACWHF r0,r2,r5      ; Dual signed fractional multiply-accumulate {r2,r3}
                      ; and r4 and return result in register r0
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
DMACWHF<.f>	a,b,c	00101bbb00111111FBBBCCCCCAAAAAA
DMACWHF<.f>	a,b,u6	00101bbb01111111FBBBuuuuuuAAAAAA
DMACWHF<.f>	b,b,s12	00101bbb10111111FBBBsssssssSSSSSS
DMACWHF<.cc><.f>	b,b,c	00101bbb11111111FBBBCCCCC0QQQQQ
DMACWHF<.cc><.f>	b,b,u6	00101bbb11111111FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMACWHU

Function

Dual unsigned 32x16 multiply and accumulate. For the non-DSP version, see [DMACWH](#).

Extension Group

MPY_OPTION == 9 or dsp_wide==true

Operation

```
if (cc) {
  accwide+=(b.w0 *cc.h0 + b.w1*c.h1)
  a=accwide
}
```



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 and B are 64-bit operands, and must be specified as even numbered registers For more information about 64-bit operands, see ["Data Formats"](#).

Instruction Format

op A,B,c

Syntax Example

DMACWHU <.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

Compute the sum of the accumulator and the two unsigned 32x16 products formed by multiplying B.w0 with c.h0, and B.w1 with c.h1. The resulting sum is assigned to the destination register pair, and is also assigned to the 64-bit accumulator (ACCH, ACCL).

The A operand must be an even-numbered register (or the null destination).

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

```
// note: accumulators modeled as 128b integers                                /* DMACWHU */
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.w0 * c.ho + b.w1 * c.h1;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = accwide & 0xffff_ffff_ffff_ffff;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff_ffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 72;
else
    rem = accwide >> 64;
if F {
    V_flag |= rem != 0;
}
```

Assembly Code Example

```
DMACWHU r0,r2,r4      ; dual unsigned 32x16 MAC of (r3, r2) and 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” and “Key for 16-bit Addressing Modes and Encoding Conventions”.

 **Note**

For detailed information about vector operands, see “Vector Operands”. For detailed information about expansion of literals, see “Expansion of Literals in Vector Instructions” and “Expansion of Literals”.

158

Instruction Code		
DMACWHU<.f>	a,b,c	00101bbb00110111FBBBCCCCC0AAAAAA
DMACWHU<.f>	a,b,u6	00101bbb01110111FBBBuuuuuuAAAAAA
DMACWHU<.f>	b,b,s12	00101bbb10110111FBBBssssssSSSSSS
DMACWHU<.cc><.f>	b,b,c	00101bbb11110111FBBBCCCCC0QQQQQ0

DMACWHU<.cc><.f> b,b,u6 00101**bbb**11**110111**F**BBB**uuuuuu**1QQQQQ**

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYH

Function

Sum of dual 16x16 multiplication. For the non-DSP version, see [DMPYH](#).

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}
```



Note

$h0$ (lower) and $h1$ (upper) represent the 16-bit elements of a 32-bit operand. $w0$ (lower) represents the 32-bit element of an operand.

Instruction Format

op a, b, c

Syntax Example

DMPYH <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set if accumulator is negative
C	☐	= Unchanged
V	☒	= Cleared

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the wide accumulator. Assign the least-significant 32 bits of the wide accumulator to the destination register.

Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

Assembly Code Example

```
DMPYH r1,r2,r3                ; Sum of dual 16x16 multiply of r2
                                ;and r3
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
DMPYH<.f>	a,b,c	00101bbb00010000FBBBCCCCCAAAAAA
DMPYH<.f>	a,b,u6	00101bbb01010000FBBBuuuuuuAAAAAA
DMPYH<.f>	b,b,s12	00101bbb10010000FBBBssssssSSSSSS
DMPYH<.cc><.f>	b,b,c	00101bbb11010000FBBBCCCCC0QQQQQ
DMPYH<.cc><.f>	b,b,u6	00101bbb11010000FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands. Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHBL

Function

Dual 16x8 signed integer multiplication of 16-bit vector elements of the first operand and the sign-extended first and second bytes of the second operand.

Extension Group

DSP Dual 16x8 MAC

Operation

$$a.h1 = b.h1 * sext(c.b1) + b.h0 * sext(c.b0);$$

Instruction Format

op a, b, c

Syntax Example

DMPYHBL <f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	=Set if the accumulator is too big to be represented in the result

Description

Perform a signed integer multiplication of the lower 16 bits of the **b** operand and the sign-extended first byte of the **c** operand. Perform a signed integer multiplication of the higher 16 bits of the **b** operand with the sign-extended second byte of the **c** operand. Add the two products and store the result in the wide accumulator. Return the least-significant 32 bits of the accumulator as the result. Flags are only updated if the **.F** suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

accwide = b.h0 * c.b0 + b.h1 * c.b1;                                /* DMPYHBL
if (DSP_CTRL.GE) // only if guard bits are enabled                */
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag = ((accwide & ~0x7fff_ffff) != 0) && ((accwide |
0x7fff_ffff) != -1);
    N_flag = accwide < 0;
}

```

Assembly Code Example

```

DMPYHBL r0,r2,r3           ; Signed integer multiplication of two vectors
                           ;r2 and r3, and add the products return result
                           ;in r0

```

Syntax and Encoding

Instruction Code		
DMPYHBL<.f>	a, b, c	00110bbb00010110FBBBCCCCC AAAAAA
DMPYHBL<.f>	a, b, u6	00110bbb01010110FBBBuuuuuu AAAAAA
DMPYHBL<.f>	b, b, s12	00110bbb10010110FBBBssssss SSSSSS
DMPYHBL<.cc><.f>	b, b, c	00110bbb11010110FBBBCCCCC 0QQQQQ
DMPYHBL<.cc><.f>	b, b, u6	00110bbb11010110FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHBM

Function

Dual 16x8 signed integer multiplication of 16-bit vector elements of the first operand and the sign-extended third and fourth bytes of the second operand.

Extension Group

DSP Dual 16x8 MAC

Operation

$$a.h1 = b.h1 * sext(c.b3) + b.h0 * sext(c.b2);$$

Instruction Format

op a, b, c

Syntax Example

DMPYHBM <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	=Set if the accumulator is too big to be represented in the result

Description

Perform a signed integer multiplication of the lower 16 bits of the *b* operand and the sign-extended third byte of the *c* operand. Signed integer multiplication of the higher 16 bits of the *b* operand with the sign-extended fourth byte of the *c* operand. Add the two products and store the result in the wide accumulator. Return the least-significant 32 bits of the accumulator as the result. Flags are only updated if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

accwide = b.h0 * c.b2 + b.h1 * c.b3;                                /* DMPYHBM
if (DSP_CTRL.GE) // only if guard bits are enabled                */
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag = ((accwide & ~0x7fff_ffff) != 0) && ((accwide |
    0x7fff_ffff) != -1);
    N_flag = accwide < 0;
}

```

Assembly Code Example

```

DMPYHBM r0,r2,r3           ; Signed integer multiplication of two vectors
                           ;r2 and r3, and add the products return result
                           ;in r0

```

Syntax and Encoding

Instruction Code

DMPYHBM<.f>	a, b, c	00110 bbb 00 010111 FBBBCCCCC AAAAAA
DMPYHBM<.f>	a, b, u6	00110 bbb 01 010111 FBBBuuuuuu AAAAAA
DMPYHBM<.f>	b, b, s12	00110 bbb 10 010111 FBBBssssss SSSSSS
DMPYHBM<.cc><.f>	b, b, c	00110 bbb 11 010111 FBBBCCCCC 0QQQQQ
DMPYHBM<.cc><.f>	b, b, u6	00110 bbb 11 010111 FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

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. $w0$ (lower) represents the 32-bit element of an operand.

Instruction Format

op a, b, c

Syntax Example

DMPYHF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if result is negative
C		= Unchanged
V	•	= Set if the accumulator value is too big to be stored in the result

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the high accumulator. Return the saturated sum of products as a 16-bit fraction. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* DMPYHF */
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(0, addlo+addhi, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
else
    // post-accumulate fractional shift mode
    addlo = b.h0 * c.h0;
    addhi = b.h1 * c.h1;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acchi = acc_add(0, addhi+addlo, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi<<1, 31, &ahi);
    else
        // guard bits are disabled
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_0000);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}

```

Assembly Code Example

```
DMPYHF r0, r1, r2          ; Signed fractional
                           ; multiplication of vectors r2 and
                           ; r1. Store the saturated result
                           ; in r0
```

Syntax and Encoding

Instruction Code

DMPYHF<.f>	a, b, c	00101 bbb 00 101010 F BBB CCCCC AAAAAA
DMPYHF<.f>	a, b, u6	00101 bbb 01 101010 F BBB uuuuuu AAAAAA
DMPYHF<.f>	b, b, s12	00101 bbb 10 101010 F BBB ssssss SSSSSS
DMPYHF<.cc><.f>	b, b, c	00101 bbb 11 101010 F BBB CCCCC 0QQQQQ
DMPYHF<.cc><.f>	b, b, u6	00101 bbb 11 101010 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHFR

Function

Sum of dual 16x16 signed fractional multiplication. The result is rounded and saturated.

Extension Group

DSP dual 16x16 MAC

Operation

```
acchi = b.h1*c.h1+b.h0*c.h0;  
a.h1 = SAT16(RND16((acchi<<1)));
```

Instruction Format

op a, b, c

Syntax Example

DMPYHFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if result is negative
C		= Unchanged
V	•	= Set if the accumulator value is too big to be stored in the result

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the high accumulator. Return the rounded and saturated sum of products as a 16-bit fraction. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(0, addlo+addhi, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi, 16), 31, &ahi);
else
    // post-accumulate fractional shift mode
    addlo = b.h0 * c.h0;
    addhi = b.h1 * c.h1;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acchi = acc_add(0, addhi+addlo, 40, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi<<1, 16), 31, &ahi);
    else
        // guard bits are disabled
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z,
&ACC0_GHI.N, &ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_0000);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}

```

Assembly Code Example

```
DMPYHFR r0, r1, r2      ; Signed fractional ;multiplication of
                        ;vectors r2 and r1. Store the saturated and
                        ;rounded result in r0
```

Syntax and Encoding

Instruction Code

DMPYHFR<.f>	a,b,c	00101 bbb 00 101011 FBBBCCCCCAAAAAA
DMPYHFR<.f>	a,b,u6	00101 bbb 01 101011 FBBBuuuuuuAAAAAA
DMPYHFR<.f>	b,b,s12	00101 bbb 10 101011 FBBBssssssSSSSSS
DMPYHFR<.cc><.f>	b,b,c	00101 bbb 11 101011 FBBBCCCCC0QQQQQ
DMPYHFR<.cc><.f>	b,b,u6	00101 bbb 11 101011 FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHU

Function

Sum of dual unsigned 16x16 multiplication. For the non-DSP version, see [DMPYHU](#).

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. `w0` (lower) represents the 32-bit element of an operand.

Instruction Format

op a, b, c

Syntax Example

DMPYHU <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Cleared

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the wide accumulator. Assign the least-significant 32 bits to the accumulator to the destination operand.

Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

accwide = b.h0 * c.h0 + b.h1 * c.h1;                                /* DMPYHU */
if (DSP_CTRL.GE) // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag = 0
}

```

Assembly Code Example

```
DMPYHU r1,r2,r3    ; sum of dual 16x16 unsigned multiply of r2 and r3
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

DMPYHU<.f>	a,b,c	00101bbb00010001FBBBCCCCCAAAAAA
DMPYHU<.f>	a,b,u6	00101bbb01010001FBBBuuuuuuAAAAAA
DMPYHU<.f>	b,b,s12	00101bbb10010001FBBBssssssSSSSSS
DMPYHU<.cc><.f>	b,b,c	00101bbb11010001FBBBCCCCC0QQQQQ
DMPYHU<.cc><.f>	b,b,u6	00101bbb11010001FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYWH

Function

Sum of dual 32x16 multiplication. For the non-DSP version, see [DMPYWH](#).

Extension Group

MPY_OPTION == 9 or dsp_wide==true

Operation

```
if (cc) {
    accwide = (b.w0 * c.h0 + b.w1 * c.h1);
    a = accwide;
}
```



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 and B are 64-bit operands, and must be specified as even numbered registers. For more information about 64-bit operands, see ["Data Formats"](#).

Instruction Format

op A, B, c

Syntax Example

DMPYWH <.f> A,B,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	= Cleared

Description

Compute the sum of the two 32x16 products formed by multiplying B.w0 with c.h0, and B.w1 with c.h1. The resulting sum is assigned to the destination register pair, and is also assigned to the 64-bit accumulator (ACCH, ACCL).

In common with all operations that produce a 64-bit result, the A operand must be an even-numbered register (or the null destination).

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

```
// note: accumulators modeled as 128b integers      /* DMPYWH */
accwide = b.w0 * c.h0 + b.w1 * c.h1;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = accwide & 0xffff_ffff_ffff_ffff;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
if F {
    N_flag = accwide < 0;
    V_flag = 0;
}
```

Assembly Code Example

```
DMPYWH r0,r2,r5    ; sum of dual 32x16 multiply of r2 and r5
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
DMPYWH<.f>	a,b,c	00101bbb00110010FBBBCCCCCAAAAAA
DMPYWH<.f>	a,b,u6	00101bbb01110010FBBBuuuuuuAAAAAA
DMPYWH<.f>	b,b,s12	00101bbb10110010FBBBsssssssSSSSSS
DMPYWH<.cc><.f>	b,b,c	00101bbb11110010FBBBCCCCC0QQQQQ
DMPYWH<.cc><.f>	b,b,u6	00101bbb11110010FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYHWF

Function

Sum of dual 16x16 signed fractional multiplication. The result is a saturated 32-bit fraction.

Extension Group

DSP dual 16x16 MAC

Operation

```
acchi = b.h1*c.h1+b.h0*c.h0;
a.h1 = SAT32(acchi<<1);
```



Note

$h0$ (lower) and $h1$ (upper) represent the 16-bit elements of a 32-bit operand. $w0$ (lower) represents the 32-bit element of an operand.

Instruction Format

op a, b, c

Syntax Example

DMPYHWF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if result is negative
C		= Unchanged
V	•	= Set if the accumulator value is too big to be stored in the result

Description

Multiply the lower 16 bits of the first and second operand, and multiply the higher 16 bits of the first and second source operands. Add the products and store the result in the high accumulator. Return the saturated sum of products as a 32-bit fraction. Flags are updated only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* DMPYHWF
s = 0;                                                                    */
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acchi = acc_add(0, addlo+addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
    else
        // guard bits disabled
        s |= sat((b.h0 * c.h0 << 1), 31, &addlo);
        s |= sat((b.h1 * c.h1 << 1), 31, &addhi);
        acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G = 0;
        ACC0_HI = acchi & 0xffff_ffff;
        v = sat(acchi, 31, &ahi);
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;
        if (DSP_CTRL.GE)
            // guard bits are enabled
            acchi = acc_add(0, addhi+addlo, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
            ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
            ACC0_GHI.G = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
            ACC0_HI = acchi & 0xffff_ffff;
            v = sat(acchi<<1, 31, &ahi);
        else
            // guard bits are disabled
            acchi = acc_add(0, addhi+addlo, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
            ACC0_GHI.G = 0;
            ACC0_HI = acchi & 0xffff_ffff;
            v = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
a = (ahi & 0xffff_ffff);
if F {
    N_flag = a < 0;
    V_flag = v;
    Z_flag = a == 0;
}

```

Assembly Code Example

```
DMPYHWF r0, r1, r2      ; Signed fractional ;multiplication of vectors r2
                        ;and r1. Store the 32-bit saturated result in r0
```

Syntax and Encoding

Instruction Code		
DMPYHWF<.f>	a,b,c	00101bbb00101000FBBBCCCCCAAAAAA
DMPYHWF<.f>	a,b,u6	00101bbb01101000FBBBuuuuuuAAAAAA
DMPYHWF<.f>	b,b,s12	00101bbb10101000FBBBssssssSSSSSS
DMPYHWF<.cc><.f>	b,b,c	00101bbb11101000FBBBCCCCC0QQQQQ
DMPYHWF<.cc><.f>	b,b,u6	00101bbb11101000FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYWHF

Function

Dual signed 32x16 fractional multiply and accumulate

Extension Group

dsp_wide==true

Operation

```
if (cc) {
    accwide += (B.w0 * c.h0 + B.w1 * c.h1) <<16;
    a = SAT32(accwide>>31);
}
```



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 and B are 64-bit operands, and must be specified as even numbered registers. For more information about 64-bit operands, see [“Data Formats”](#).

Instruction Format

op a, B, c

Syntax Example

DMPYWHF <.f> a,B,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	= Set if the result saturates

Description

Dual signed fractional multiply 2x32b operand B and 2x16b operand C; add the products to the wide accumulator. Return the saturated most significant 32b word 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 only be updated if the .F suffix is used.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* DMPYWHF
s = 0;                                                                           */
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add0 = b.w0 * c.h0 << 17;
        add1 = b.w1 * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add0 + add1, 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.w0 * c.h0 << 17), 63, &add0) | sat((b.w1 * c.h1 << 17), 63,
&add0);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add0 + add1, 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.w0 * c.h0 + b.w1 * 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.w0 * c.h0 + b.w1 * 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

```
DMPYWHF r0,r2,r5    ; sum of dual 32x16 multiply of r2 and r5
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
DMPYWHF<.f>	a, b, c	00110bbb00111101FBBBCCCCCAAAAA
DMPYWHF<.f>	a, b, u6	00110bbb01111101FBBBuuuuuuAAAAA
DMPYWHF<.f>	b, b, s12	00110bbb10111101FBBBsssssssSSSSSS
DMPYWHF<.cc><.f>	b, b, c	00110bbb11111101FBBBCCCCC0QQQQQ
DMPYWHF<.cc><.f>	b, b, u6	00110bbb11111101FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

DMPYWHU

Function

Sum of dual unsigned 32x16 multiplication. For the non-DSP version, see [DMPYWHU](#).

Extension Group

MPY_OPTION == 9 or dsp_wide==true

Operation

```
if (cc) {
    accwide = (b.w0 * c.h0 + b.w1 * c.h1);
    a = accwide;
}
```



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 and B are 64-bit operands, and should be specified as even numbered registers. For more information about 64-bit operands, see ["Data Formats"](#).

Instruction Format

op A,B,c

Syntax Example

DMPYWHU <.f> A,B,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Cleared

Description

Compute the sum of the two unsigned 32x16 products formed by multiplying B.w0 with c.h0, and B.w1 with c.h1. The resulting sum is assigned to the destination register pair, and is also assigned to the 64-bit accumulator (ACCH, ACCL).

In common with all operations that produce a 64-bit result, the A operand must be an even-numbered register (or the null destination).

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is always cleared.

Pseudo Code

```
accwide = b.w0 * c.h0 + b.w1 * c.h1;          /* DMPYWHU */
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = accwide & 0xffff_ffff_ffff_ffff;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
if F {
    V_flag = 0;
}
```

Assembly Code Example

```
DMPYWHU r0,r2,r5    ; sum of dual unsigned 32x16 multiplication of (r3,r2)
                   ;and r5. The result is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
DMPYWHU<.f>	a,b,c	00101bbb00110011FBBBCCCCCAAAAAA
DMPYWHU<.f>	a,b,u6	00101bbb01110011FBBBuuuuuuAAAAAA
DMPYWHU<.f>	b,b,s12	00101bbb10110011FBBBsssssssSSSSSS
DMPYWHU<.cc><.f>	b,b,c	00101bbb11110011FBBBCCCCC0QQQQQ
DMPYWHU<.cc><.f>	b,b,u6	00101bbb11110011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

FLAGACC

Function

Copy accumulator flags to the STATUS32 register.

Extension Group

DSP Accumulator

Operation

```
switch (c.b1) {
0,2,3: STATUS32.ZNCV = ACC0_GHI.ZNCV;
1: STATUS32.ZNCV = ACC0_GLO.ZNCV;
}
```

Instruction Format

op c

Syntax Example

FLAGACC c

STATUS32 Flags Affected

Z	•	= From accumulator
N	•	= From accumulator
C	•	= From accumulator
V	•	= From accumulator

Description

Copy flags from the accumulator guard register to the STATUS32 register. The *c* operand consists of one field: the accumulator select.

Using the loop-counter (LP_COUNT/r60) as a source operand to the FLAGACC instruction results in an illegal instruction exception.

Pseudo Code

```
switch (c.b1) {                                     /* FLAGACC */
  case 0, 2, 3:
    STATUS32.ZNCV = ACC0_GHI.ZNCV;
    break;
  case 1:
    STATUS32.ZNCV = ACC0_GLO.ZNCV;
    break;
}
```

Assembly Code Example

```
FLAGACC r2      ; Get accumulator flags information from the fields in r2
```

Syntax and Encoding

Instruction Code		
FLAGACC	c	00101100001011111000CCCCC111111
FLAGACC	u6	00101100011011111000uuuuuu111111

GETACC

Function

Get the accumulator value

Extension Group

DSP Accumulator

Operation

```
b = shift_round_sat(acc, c);
```

Instruction Format

op b,c

Syntax Example

GETACC b, c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Arithmetic left shift of the accumulator including the guard bits; optionally round and saturate the result. The result can be MSB or LSB aligned.

If, `dsp_accshift==full`, the shift amount is saturated to a range of $-72 \leq \text{shamt} \leq +72$; if `dsp_accshift==limited`, the shift amount is saturated to a range of $-8 \leq \text{shamt} \leq +8$.

Example 1: Cast the wide accumulator (`DSP_CTRL.PA = 1`) to Q8 while saturating and rounding:

```
GETACC %r0, 24|0<<8|1<<10|1<<11|2<<12|3<<14
```

Example 2: Cast the high and low accumulator (`DSP_CTRL.PA = 0`) to a 2-way Q15 vector after shifting left by 2 while saturating and truncating:

```
GETACC %r0, (16+3)|3<<8|1<<10|1<<11|1<<12|2<<14 ; note 16+3 because post-accumulate ;shift-mode and shift left 2
```

Example 3: Cast the wide accumulator to an integer.

```
GETACC %r0, 32|0<<8|0<<10|1<<11|0<<12|0<<14
```

Instruction Encoding

Operand c	Description
Bits [7:0]	Shift amount; an 8 bit signed left-shift amount for reading the accumulator
Bits [9:8]	Accumulator select. 0: selects the wide accumulator; 1: selects the low accumulator; 2: selects the high accumulator; 3: selects dual hi/lo accumulator output.
Bit [10]	Enables saturation
Bit [11]	1: use arithmetic mode
Bit [12]	1: enable rounding; rounding is controlled by the DSP_CTRL.RM bit 0: enable truncation
Bit [13]	LSB/MSB alignment 0: MSB alignment 1: LSB alignment
Bit [15:14]	Rounding bit 0: result is 32 bits (rounding is at bit -1); 1: result is 24 bits rounding occurs at bit 7; 2: result is 16 bits and rounding occurs at bit 15; 3: result is 8b and rounding happen at bit 23. After rounding the result always resides in the reminaing MSBs, LSBs is set to 0.
Bit [16]	The GETACC instruction is sensitive to the PA mode bit; output data is shifted by one additional bit

Pseudo Code

```

// this code uses accumulator representation of 256b precision
if (DSP_ACCSHIFT == DSP_DSP_SHIFT_FULL)
    shamt = c.b0 < -72 ? -72 : (c.b0 > 72 ? 72 : c.b0);
else
    shamt = c.b0 < -8 ? -8 : (c.b0 > 8 ? 8 : c.b0);
accsel = C & 0x0000_0300 >> 8; // accumulator select:
                                // 0: wide accumulator
                                // 1: low accumulator
                                // 2: high accumulator
                                // 32: dual accumulator (returns 2x16b vector)
sat = C & 0x0000_0400 >> 10; // saturation enable
signed = C & 0x0000_0800 >> 11; // signed operation
rnd_en = C & 0x0000_1000 >> 12; // round mode
lsb = C & 0x0000_2000 >> 13; // LSB/MSB alignment
rndbyte = C & 0x0000_c000 >> 14; // index of byte to round
pa = C & 0x0001_0000 >> 16; // PA bit sensitive, one bit extra shift
if (rnd_en == 1)
    rndmode = DSP_CTRL.RM;
else
    rndmode = 0;
if (DSP_CTRL.GE)
    accwide = ACC0_GHI.GUARD << 64 | unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO);
    acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
    acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
else
    accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
    acchi = ACC0_HI;
    acclo = ACC0_LO;
if (accsel == 0)
    hi = accwide;
else
    hi = acchi << 32;
lo = acclo << 32;
if (pa && !DSP_CTRL.PA)
    // one extra bit left shift for post-accumulate shift mode
    shamt++;
// sign extend on left, zero extend on right
hiext = signed ? sext(hi) << 72 : ext(hi) << 72;
loext = signed ? sext(lo) << 72 : ext(lo) << 72;
// shift
shamt = shamt < -72 ? -72 : (shamt > 72 ? 72 : shamt);
hishift = shamt >= 0 ? hiext << shamt : hiext >> -shamt;
loshift = shamt >= 0 ? loext << shamt : loext >> -shamt;
// round at dot -8, -16, -24, -32
hirnd = round(hishift, 72+32+8*rndbyte, rndmode)>>72;
lornd = round(loshift, 72+32+8*rndbyte, rndmode)>>72;

```

```
// saturate at fractional dot
if (sat) {
    sh = sat(hirnd, 63, &hisat); // signed or unsigned saturation
    sl = sat(lornd, 63, &losat);
} else {
    hisat = hirnd;
    losat = lornd;
}
// mask of the LSBs that got rounded
switch (rndbyte) {
    3: hisat &= 0xff000000_00000000; losat &= 0xff000000_00000000;
    2: hisat &= 0xffff0000_00000000; losat &= 0xffff0000_00000000;
    1: hisat &= 0xffffff00_00000000; losat &= 0xffffff00_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	00101110001011110111CCCCC011000
GETACC	0,limm	00101110001011110111111110011000

GETACC	b,u6	00101bbb011011110BBBuuuuuu011000
GETACC	0,u6	00101110011011110111uuuuuu011000

MAC

Function

Signed 32x32 multiplication and accumulation. For the non-DSP version, see [MAC](#).

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    a = result.w0;
    acc = result;
}
```

Instruction Format

op a, b, c

Syntax Example

MAC <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set if the accumulator is negative
C	☐	= Unchanged
V	☒	= Set if an overflow occurs during accumulation. This instruction never clears this flag.

Description

Multiply the 32 bits of the first and second source operands to get a 64-bit product. This 64-bit product is added to the wide accumulator to form the result. Return the least-significant 32 bits of the wide accumulator in the destination register.

Flags are updated only if the set-flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
else
    accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
accwide = accwide_org + b * c;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
a = (accwide & 0xffff_ffff);
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    N_flag = rem & 1;
    V_flag |= rem != -1 && rem != 0;
}
```

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	00101bbb000001110FBBBCCCCC(AAAAAA)
MAC<.f>	a,b,u6	00101bbb01001110FBBBuuuuuu(AAAAAA)
MAC<.f>	b,b,s12	00101bbb10001110FBBBssssss(SSSSSS)
MAC<.cc><.f>	b,b,c	00101bbb11001110FBBBCCCCC(0QQQQQ)
MAC<.cc><.f>	b,b,u6	00101bbb11001110FBBBuuuuuu(1QQQQQ)

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACD

Function

Signed 32x32 multiplication and accumulation. Returns a 64-bit result. For the non-DSP version, see [MACD](#).

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    A = result;
    acc = result;
}
```

Instruction Format

op A, b, c

Syntax Example

MACD <.f> A,b,c

Timing Characteristics

Single cycle

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	= Set if an overflow occurs during accumulation. This instruction never clears this flag.

Description

Multiply the 32 bits of the first and second source operands to get a 64-bit product. This 64-bit product is added to the wide accumulator to form the result. Return the least-significant 64 bits of the wide accumulator in the destination register.

Flags are updated only if the set-flags suffix (.F) is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (LP_COUNT/r60) as a destination operand register.

- Using an odd destination register for DSP instructions with 64-bit 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_ffff_ffff);
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    N_flag = rem & 1;
    V_flag |= rem != -1 && rem != 0;
}
```

Assembly Code Example

```
MACD r0,r2,r3      ; 32x32y64 multiply-accumulate r2 and r3 and store
                   ;result in (r1,r0)
```

Syntax and Encoding

Instruction Code

MACD<.f>	a,b,c	00101bbb00011010FBBBCCCCC0AAAAAA
MACD<.f>	a,b,u6	00101bbb01011010FBBBuuuuuuAAAAAA
MACD<.f>	b,b,s12	00101bbb10011010FBBBssssssSSSSSS
MACD<.cc><.f>	b,b,c	00101bbb11011010FBBBCCCCC00QQQQQ
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:

`SUB.F 0, R0, R1`

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACDF

Function

32x32 signed fractional multiplication and accumulation.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide += b * c;
    A = SAT64(accwide<<1);
}
```



Note

A is a register pair representing a 64-bit operand.

Instruction Format

op A, b, c

Syntax Example

MACDF <.f> A,b,c

Timing Characteristics

Two cycle

STATUS32 Flags Affected

Z		• = Set if the result is zero
N		• = Set to the resulting sign is negative
C		= Unchanged
V		• = Set if the result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands. The product is added to the wide accumulator to form the result. The saturated least-significant 64-bits of the result is then assigned to the destination register pair.

If the result saturates, the sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix. Flags are updated only if the set-flags suffix (`.F`) is used.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit 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	00110bbb00010011FBBBCCCCCAAAAAA
MACDF<.f>	a,b,u6	00110bbb01010011FBBBuuuuuuAAAAAA
MACDF<.f>	b,b,s12	00110bbb10010011FBBBssssssSSSSSS
MACDF<.cc><.f>	b,b,c	00110bbb11010011FBBBCCCCC0QQQQQ
MACDF<.cc><.f>	b,b,u6	00110bbb11010011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACDU

Function

Unsigned 32x32 multiplication and accumulation. Return a 64-bit result. For the non-DSP version, see [MACDU](#).

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = acc + unsigned(b * c);
    A = result;
    acc = result;
}
```

Instruction Format

op A, b, c

Syntax Example

MACDU <.f> A,b,c

Timing Characteristics

Two cycle

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V	•	= Set if the accumulator overflows. This instruction never clears this flag.

Description

Multiply the 32 bits of the first and second source operands to get a 64-bit product. This 64-bit product is added to the wide accumulator to form the result. Return the least-significant 64 bits of the wide accumulator in the destination register.

Flags are updated only if the set-flags suffix (.F) is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (LP_COUNT/r60) as a destination operand register.

- Using an odd destination register for DSP instructions with 64-bit 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

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
MACDU<.f>	a,b,c	00101bbb000111011FBBBCCCCC#####
MACDU<.f>	a,b,u6	00101bbb010111011FBBBuuuuuu#####
MACDU<.f>	b,b,s12	00101bbb100111011FBBBssssssSSSSSS
MACDU<.cc><.f>	b,b,c	00101bbb110111011FBBBCCCCC0#####
MACDU<.cc><.f>	b,b,u6	00101bbb110111011FBBBuuuuuu1#####

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

`SUB.F 0, R0, R1`

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACF

Function

Signed 32x32 fractional multiplication and accumulation. Return a saturated 32-bit result.

Extension Group

DSP 32x32 MAC

Operation

```
if (cc) {
    accwide += b * c;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set to the resulting sign is negative
C		= Unchanged
V	•	= Set if the result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands. The product is added to the wide accumulator to form the result. The truncated and saturated second 32-bit word of the result is then assigned to the destination register.

The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. Flags are updated only if the set-flags suffix (`.F`) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* MACF*/
s= 0;
if (DSP_CTRL.PA)
  if (DSP_CTRL.GE)
    add = b * c << 1;
    accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
    ACC0_GHI.V = sat_en(accwide_org + add, 71, &accwide);
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
    ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
  else
    s = sat((b * c << 1), 63, &add);
    accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
    ACC0_GHI.V = sat(accwide_org + add, 63, &accwide);
    ACC0_GHI.G = 0;
    v = sat(accwide >> 32, 31, &a);
  else
    if (DSP_CTRL.GE)
      accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
      ACC0_GHI.V = sat_en(accwide_org + b * c, 71, &accwide);
      ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
      ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
      accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
      ACC0_GHI.V = sat(accwide_org + b * c, 63, &accwide);
      ACC0_GHI.G = 0;
      v = sat(accwide >> 31, 31, &a);
    DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
    ACC0_HI = (accwide >> 32) & 0xffff_ffff;
    ACC0_LO = accwide & 0xffff_ffff;
    ACC0_GHI.Z = accwide == 0;
    ACC0_GHI.N = accwide < 0;
    if F {
      Z_flag = a = 0;
      N_flag = a < 0;
      V_flag = v;
    }

```

Assembly Code Example

```

MACF r0,r1,r2      ; signed fractional 32x32 multiply-accumulate r1 and r2
                   ;and store the saturated result in r0

```

Syntax and Encoding

		Instruction Code
MACF<.f>	a, b, c	00110 bbb 00 001100 F BBB CCCCC AAAAAA
MACF<.f>	a, b, u6	00110 bbb 01 001100 F BBB uuuuuu AAAAAA
MACF<.f>	b, b, s12	00110 bbb 10 001100 F BBB ssssss SSSSSS
MACF<.cc><.f>	b, b, c	00110 bbb 11 001100 F BBB CCCCC 0QQQQQ
MACF<.cc><.f>	b, b, u6	00110 bbb 11 001100 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACFR

Function

Signed 32x32 fractional multiplication and accumulation. Return a saturated and rounded 32-bit result.

Extension Group

DSP 32x32 MAC

Operation

```
if (cc) {
    accwide += b * c;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set to the resulting sign is negative
C		= Unchanged
V	•	= Set if the result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands. The product is added to the wide accumulator to form the result. The rounded and saturated higher 32 bits of the result is then assigned to the destination register.

The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. Flags are updated only if the set-flags suffix (`.F`) is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MACFR*/
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI)
        << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide_org + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD !=
-1 : ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide_org + add, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(round(accwide,32) >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide_org = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI)
        << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide_org + b * c, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD !=
-1 : ACC0_GHI.GUARD != 0;
    else
        accwide_org = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide_org + b * c, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(round(accwide,31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MACFR r0,r1,r2    ; signed fractional 32x32 multiply-accumulate r1 and r2
                  ;and store the saturated result in r0

```

Syntax and Encoding

		Instruction Code
MACFR<.f>	a,b,c	00110bbb00001101FBBBCCCCC AAAAAA
MACFR<.f>	a,b,u6	00110bbb01001101FBBBuuuuuu AAAAAA
MACFR<.f>	b,b,s12	00110bbb10001101FBBBssssss SSSSSS
MACFR<.cc><.f>	b,b,c	00110bbb11001101FBBBCCCCC 0QQQQQ
MACFR<.cc><.f>	b,b,u6	00110bbb11001101FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACU

Function

Unsigned 32x32 multiplication and accumulation. For the non-DSP version, see [MACU](#).

Extension Group

MPY_OPTION > 6 or has_dsp==true

Operation

```
if (cc) {
    result = acc + (b * c);
    a = result.w0;
    acc = result;
}
```



Note

w0 represents the lower 32-bits of a 64-bit result.

Instruction Format

op a, b, c

Syntax Example

MACU <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Set if the accumulator overflows. This instruction never clears this flag.

Description

Perform an unsigned multiplication of the 32 bits of the first and second source operands to get a 64-bit product. This 64-bit product is added to the wide accumulator to form the result. Return the least-significant 32 bits of the wide accumulator in the destination register.

Flags are updated only if the set-flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MACU*/
if (DSP_CTRL.GE)
    accwide_org = unsigned(ACC0_GHI.GUARD) << 64 |
    unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO)
else
    accwide_org = unsigned(ACC0_HI) << 32 | unsigned(ACC0_LO)
accwide = accwide_org + b * c;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 72
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
else
    rem = accwide >> 64
a = (accwide & 0xffff_ffff);
if F {
    V_flag |= rem != 0;
}

```

Assembly Code Example

```

MACU r1,r2,r3                                ; 32x32 unsigned multiplication
                                              ;and accumulation of r2 and r3
                                              ;and store result in r1

```

Syntax and Encoding

Instruction Code

MACU<.f>	a,b,c	00101bbb000001111FBBBCCCCC	AAAAAA
MACU<.f>	a,b,u6	00101bbb010001111FBBBuuuuuu	AAAAAA
MACU<.f>	b,b,s12	00101bbb100001111FBBBssssss	SSSSSS
MACU<.cc><.f>	b,b,c	00101bbb110001111FBBBCCCCC	0QQQQQ
MACU<.cc><.f>	b,b,u6	00101bbb110001111FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHFM

Function

Signed 32x16 fractional multiplication and accumulation.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h1 <<32;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHFM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the saturated most-significant 32 bits of the accumulator, without guard bits. The sticky `DSP_CTRL.SAT` flag is set if saturation occurs, regardless of the `.F` suffix. Other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MACWHFM
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

```
MACWHFM r0,r2,r3                ; Signed fractional multiply-accumulate r2 and
                                ;r3 and return result in register r0
```

Syntax and Encoding

		Instruction Code
MACWHFM<.f>	a, b, c	00110bbb00100010FBBBCCCCC AAAAAA
MACWHFM<.f>	a, b, u6	00110bbb01100010FBBBuuuuuu AAAAAA
MACWHFM<.f>	b, b, s12	00110bbb10100010FBBBssssss SSSSSS
MACWHFM<.cc><.f>	b, b, c	00110bbb11100010FBBBCCCCC 0QQQQQ
MACWHFM<.cc><.f>	b, b, u6	00110bbb11100010FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHFMR

Function

Signed 32x16 fractional multiplication and accumulation. The result is rounded and saturated.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h1 <<32;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHFMR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the rounded and saturated most-significant 32 bits of the accumulator without guard bits. The sticky `DSP_CTRL.SAT` flag is set if saturation occurs, regardless of the `.F` suffix. Other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            ACC0_GHI.V = sat_en(accwide + b * c.h1 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            ACC0_GHI.V = sat(accwide + b * c.h1 << 16, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(round(accwide, 31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}
```

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	00110bbb00100011FBBBCCCCCAAAAA
MACWHFMR<.f>	a, b, u6	00110bbb01100011FBBBuuuuuuAAAAA
MACWHFMR<.f>	b, b, s12	00110bbb10100011FBBBssssssSSSSSS
MACWHFMR<.cc><.f>	b, b, c	00110bbb11100011FBBBCCCCC0QQQQQ
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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHL

Function

Signed 32x16 integer multiplication and accumulation.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h0;
    a = accwide;
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHL <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	=Set if overflow occurs during accumulation; this instruction never clears this flag

Description

Signed integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the least-significant 32 bits of the accumulator. Flags are updated only if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
    accwide += b * c.h0;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
else
    accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
    accwide += b * c.h0;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 71;
else
    rem = accwide >> 63;
if F {
    N_flag = accwide < 0;
    V_flag |= rem != -1 && rem != 0;
}
```

Assembly Code Example

```
MACWHL r0,r2,r3      ; Signed integer multiply-accumulate r2 and r3
                     ;and return result in register r0
```

Syntax and Encoding

Instruction Code

MACWHL<.f>	a, b, c	00110bbb00011101FBBBCCCCC
MACWHL<.f>	a, b, u6	00110bbb01011101FBBBuuuuuu
MACWHL<.f>	b, b, s12	00110bbb10011101FBBBssssss
MACWHL<.cc><.f>	b, b, c	00110bbb11011101FBBBCCCCC0
MACWHL<.cc><.f>	b, b, u6	00110bbb11011101FBBBuuuuuu1

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHFL

Function

Signed 32x16 fractional multiplication and accumulation.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h0 <<32;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHFL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the `b` operand and the least-significant 16 bits of the `c` operand. Add the product to the wide accumulator. Return the saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the `.F` suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the `.F` suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* MACWHFL */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            ACC0_GHI.V = sat_en(accwide + b * c.h0 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            ACC0_GHI.V = sat(accwide + b * c.h0 << 16, 63, &accwide); v =
sat(accwide>>31, 31, &a);
            ACC0_GHI.G = 0;
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }

```

Assembly Code Example

```

MACWHFL r0,r2,r3          ; Signed fractional multiply-accumulate r2 and r3
                           ;and return result in register r0

```


Syntax and Encoding

		Instruction Code
MACWHFL<.f>	a, b, c	00110bbb00100110FBBBCCCCCAAAAA
MACWHFL<.f>	a, b, u6	00110bbb01100110FBBBuuuuuuAAAAA
MACWHFL<.f>	b, b, s12	00110bbb10100110FBBBssssssSSSSSS
MACWHFL<.cc><.f>	b, b, c	00110bbb11100110FBBBCCCCC0QQQQQ
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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHFLR

Function

Signed 32x16 fractional multiplication and accumulation with rounding.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += b * c.h0 << 32;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHFLR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the rounded and saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* MACWHFLR */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide + b * c.h0 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide + b * c.h0 << 16, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide, 31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MACWHFLR r0,r2,r3          ; Signed fractional multiply-accumulate r2 and r3 and
                           ; return the rounded and saturated result in register r0

```

Syntax and Encoding

		Instruction Code
MACWHFLR<.f>	a, b, c	00110bbb00100111FBBBCCCCC AAAAAA
MACWHFLR<.f>	a, b, u6	00110bbb01100111FBBBuuuuuu AAAAAA
MACWHFLR<.f>	b, b, s12	00110bbb10100111FBBBssssss SSSSSS
MACWHFLR<.cc><.f>	b, b, c	00110bbb11100111FBBBCCCCC 0QQQQQ
MACWHFLR<.cc><.f>	b, b, u6	00110bbb11100111FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHKL

Function

Signed 32x16 integer multiplication and accumulation and shift-right the lower bits

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    acchi += b * c.h0 >> 16;
    a = acchi;
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHKL <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	=Set if the accumulator overflows; this instruction never clears this flag

Description

Signed integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and add the shifted product to the high accumulator. Return the least-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MACWHKL */
if (DSP_CTRL.GE)
    acchi = sext(ACC0_GHI.GUARD)<<32 | unsigned(ACC0_HI)
    acchi += (b * c.h0) >> 16;
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
else
    acchi = ACC0_HI;
    acchi += (b * c.h0) >> 16;
ACC0_HI = acchi & 0xffff_ffff;
a = acchi & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = acchi >> 39;
else
    rem = acchi >> 31;
if F {
    N_flag = rem & 1;
    V_flag |= rem != -1 && rem != 0;
}
```

Assembly Code Example

```
MACWHKL r0,r2,r3                ; Signed integer multiply-accumulate r2 and r3
                                ; shift result by 16 bits, and return the shifted
                                ; result in register r0
```

Syntax and Encoding

Instruction Code

MACWHKL<.f>	a,b,c	00110bbb00101000FBBBCCCCC	AAAAAA
MACWHKL<.f>	a,b,u6	00110bbb01101000FBBBuuuuuu	AAAAAA
MACWHKL<.f>	b,b,s12	00110bbb10101000FBBBssssss	SSSSSS
MACWHKL<.cc><.f>	b,b,c	00110bbb11101000FBBBCCCCC	0QQQQQ
MACWHKL<.cc><.f>	b,b,u6	00110bbb11101000FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHKUL

Function

Unsigned 32x16 integer multiplication and accumulation and shift-right the lower bits

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    acchi += b * c.h0 >> 16;
    a = acchi;
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHKUL <.f> a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	=Set if the accumulator overflows; this instruction never clears this flag

Description

Unsigned integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and add the shifted product to the high accumulator. Return the least-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used.

Pseudo Code

```

if (DSP_CTRL.GE)                                     /* MACWHKUL */
    acchi = unsigned(ACC0_GHI.GUARD)<<32 | unsigned(ACC0_HI);
    acchi += (b * c.h0) >> 16;
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    s = acchi >> 40 != 0;
else
    acchi = unsigned(ACC0_HI);
    acchi += (b * c.h0) >> 16;
    s = acchi >> 32 != 0;
ACC0_HI = acchi & 0xffff_ffff;
a = acchi & 0xffff_ffff;
if F {
    V_flag |= s;
}

```

Assembly Code Example

```

MACWHKUL r0,r2,r3      ; Unsigned integer multiply-accumulate r2 and r3
                        ; shift result by 16 bits, and return the shifted
                        ; result in register r0

```

Syntax and Encoding

Instruction Code

MACWHKUL<.f>	a, b, c	00110bbb00101001FBBBCCCCC AAAAAA
MACWHKUL<.f>	a, b, u6	00110bbb01101001FBBBuuuuuu AAAAAA
MACWHKUL<.f>	b, b, s12	00110bbb10101001FBBBssssss SSSSSS
MACWHKUL<.cc><.f>	b, b, c	00110bbb11101001FBBBCCCCC 0QQQQQ
MACWHKUL<.cc><.f>	b, b, u6	00110bbb11101001FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MACWHUL

Function

Unsigned 32x16 integer multiplication and accumulation.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide += (unsigned) (b * c.h0);
    a = accwide;
}
```

Instruction Format

op a, b, c

Syntax Example

MACWHUL <.f> a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Unchanged
C		= Unchanged
V	•	=Set if the accumulator overflows.; sticky

Description

Unsigned integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Add the product to the wide accumulator. Return the least-significant 32 bits of the accumulator. Flags are updated only if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide = unsigned(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
    accwide += b * c.h0;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
    s = accwide >> 72 != 0;
else
    accwide = unsigned(ACC0_HI << 32) | unsigned(ACC0_LO);
    accwide += b * c.h0;
    s = accwide >> 64 != 0;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    V_flag |= s;
}
/* MACWHUL
*/

```

Assembly Code Example

```

MACWHUL r0,r2,r3      ; Unsigned integer multiply-accumulate r2 and r3
                      ;and return result in register r0

```

Syntax and Encoding

Instruction Code

MACWHUL<.f>	a, b, c	00110bbb00011111FBBBCCCCC AAAAAA
MACWHUL<.f>	a, b, u6	00110bbb01011111FBBBuuuuuu AAAAAA
MACWHUL<.f>	b, b, s12	00110bbb10011111FBBBssssss SSSSSS
MACWHUL<.cc><.f>	b, b, c	00110bbb11011111FBBBCCCCC 0QQQQQ
MACWHUL<.cc><.f>	b, b, u6	00110bbb11011111FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MODIF

Function

Update an address pointer based on an operand modifier value

Extension Group

AGU

Operation

```
AGU_AUX_AP[c.ptr_reg] = next_addr(c);
```

Instruction Format

op c

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 $2^{VW} < \text{container size}$, 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 = unnumber of bits in
address
    r = (r << 1) | (a & 1);
    a >>= 1;
}
return r;
}
uint32 log2up(uint32 a) {
// find next power of 2 bigger than or equal to a
uint32 r = 0;
while ((1 << r) < a) r++;
return r;
}
void apply_modifier(mod_t mod) { // the modifier
uint32 addr; // current address
uint32 addr_nxt; // next address
int32 offset; // signed offset
int32 wrap; // unsigned wrap boundary, minus 1
int32 mask; // mask for wrapping
// retrieve pointer value
addr = AGU_AUX_AP[mod.ptr_reg];
// get offset
if ((mod.opc % 2) == 0) {
    offset = AGU_AUX_OS[mod.offset_reg];
} else {
    offset = unsigned(mod.offset_imm);
}
// get wrap boundary or scale offset
if (mod.opc == 4) {
    wrap = AGU_AUX_OS[mod.wrap_reg];
} else if (mod.opc == 5 || mod.opc == 6) {
    wrap = unsigned(power(2, mod.wrap_imm+1))-1;
} else {
    offset = offset << mod.sc;
}
}

```



```
// flip pointer and offset bits if bit-reverse
if (mod.opc == 2 || mod.opc == 3) {
    addr    = bitrev(addr);
    offset  = bitrev(offset);
}
// optionally negate offset
if (mod.dir)
    offset = -offset;
addr_nxt = addr + offset;
if (mod.opc == 2 || mod.opc == 3) {
    // flip back pointer bits if bit-reverse, avoid carry prop into lsbs
    addr_nxt = bitrev(addr_nxt); // bitrev is flipping bits in 32b word
    lsbmask = -power(2, mod.vw-mod.cs+mod.ds); // create a mask to clear
lsbs
    addr_nxt &= lsbmask;
} else if (mod.opc >= 4) {
    // wrap
    mask = power(2, log2up(wrap+1))-1; // log2up is base 2 logarithm
rounded up
    if (((addr_nxt & ~mask) != (addr & ~mask)) ||
        ((addr_nxt & mask) > wrap)) {
        // went through the wrap boundary
        if (offset > 0) {
            addr_nxt -= wrap - 1; // went over max
        } else {
            addr_nxt += wrap + 1; // went under min
        }
    }
} // else transparent
// update address with next value
AGU_AUX_AP[mod.ptr_reg] = addr_nxt;
}
```

Assembly Code Example

```
MODIF c          ; Update the address pointer based on the immediate
                  ; modifier value
```

Syntax and Encoding

		Instruction Code
MODIF	c	00101101001011110000CCCCC111111
MODIF	u6	00101101011011110000uuuuuu111111

MODAPP

Function

Apply a modifier without accessing the memory

Extension Group

AGU

Operation

```
if (c == agu_ux) {
    AGU_AUX_AP[c.ptr_reg] = next_addr(c);
}
if (b == agu_ux) {
    AGU_AUX_AP[b.ptr_reg] = next_addr(b);
}
```

Instruction Format

op b, c

Syntax Example

MODAPP b, c

MODAPP 0, c

Timing Characteristics

Issue one instruction every two cycles

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

This instruction takes two modifiers as input. If the operands are AGU window registers, that is, $32 \leq B < \text{AGU\_MOD\_NUM} + 32$, $32 \leq C < \text{AGU\_MOD\_NUM} + 32$, the associated modifier is applied to update the address pointer. If an operand is a non AGU window register or if the operand is an immediate, the address pointer is not updated. This instruction does not return a value or update the flags.

This instruction does not raise any exception with an illegal modifier or an illegal address. However, this instruction implements the following checks:

- If an operand refers to a non-existing modifier register (AGU_AUX_MODx), the associated address pointer is not updated.
- If an operand refers to a non-existing address pointer (AGU_AUX_APx), the associated address pointer is not updated.
- If an operand refers to a non-existing offset register (AGU_AUX_OSx), the associated address pointer is not updated.
- If an operand refers to an illegal Fx ($AGU\_AUX\_MODx[10:6]$) value, the associated address pointer is not updated. Illegal Fx values are 15 and greater than 22.
- If a source operand uses $AGU\_AUX\_MODx[5:4] == 3$, the associated address pointer is not updated.
- If a source operand uses $2^{VW} < \text{container size}$, the associated address pointer is not updated. Where, VW is the $AGU\_AUX\_MODx[5:4] == 3$ field.

Pseudo Code

```
// BX is the index of the B operand register
if (BX >= 32 && BX <= AGU_MOD_NUM+32) {
    m = AGU_AUX_MOD[BX-32];
    apply_modifier(m);
}
// CX is the index of the C operand register
if (CX >= 32 && CX <= AGU_MOD_NUM+32) {
    n = AGU_AUX_MOD[CX-32];
    apply_modifier(n);
}
```

/*
MODAPP*/

Assembly Code Example

```
MODAPP agu_u2, agu_u3      ;Apply the second and third modifier
                           ;updating the associated address pointer
```

Syntax and Encoding

Instruction Code

MODAPP	b, c	00101bbb001011110BBBCCCCC111110
MODAPP	b, u6	00101bbb011011110BBBuuuuuu111110

MPY

Function

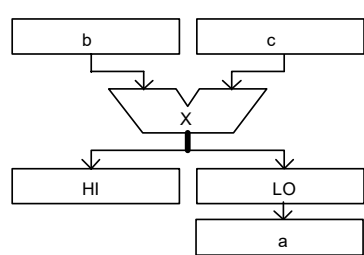
32 x 32 Signed Multiply, returning least-significant 32-bit word of the result. For the non-DSP version, see [MPY](#), [MPY_S](#).

Extension Group

MPY_OPTION > 1 or has_dsp==true

Operation

if (cc) a = (signed) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPY<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 64-bit result is set
C		= Unchanged

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers
t  = b * c;
a  = (t & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = t < 0;
    V_flag = (t >> 31) != 0 && (accwide >> 31) != -1;
}
```

```
/* MPY */
/*when has_dsp==true
```

Assembly Code Example

```
MPY r1,r2,r3      ; Multiply r2 by r3
MPY_S r1,r2        ; and put the least-significant
                   ; word of the result in r1
                   ; Multiply r1 by r2 and put low
                   ; part of result in r1
```

Syntax and Encoding

Instruction Code

MPY<.f>	a,b,c	00100bbb00011010FBBBCCCCC
MPY<.f>	a,b,u6	00100bbb01011010FBBBuuuuuu
MPY<.f>	b,b,s12	00100bbb10011010FBBBssssss
MPY<.cc><.f>	b,b,c	00100bbb11011010FBBBCCCCC0
MPY<.cc><.f>	b,b,u6	00100bbb11011010FBBBuuuuuu1
MPY_S	b,b,c	01111bbbccc01100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPY_S

Function

Signed 32 x 32 multiplication. Return the least-significant 32 bits of the result. For the non-DSP version, see [MPY](#), [MPY_S](#).

Extension Group

MPY_OPTION > 1 or has_dsp==true

Operation

$t = b * c;$

$b = t.w0;$

Instruction Format

op b, b, c

Syntax Example

MPY_S b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Return the least-significant 32 bits of the product in the destination register. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers          /* MPY_S */
t  = b * c;
b  = (t & 0xffff_ffff);
```

Assembly Code Example

```
MPY_S r1,r2                ; Signed multiplication of r1 an
                             ;r2, and return the result in r2
```

Syntax and Encoding

Instruction Code		
MPY_S	b,c	01111bbbccc01100

MPYD

Function

Signed 32x32 multiplication. Return a 64-bit result. For the non-DSP version, see [MPYD](#).

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```

Instruction Format

op A, b, c

Syntax Example

MPYD <.f> A,b,c

STATUS32 Flags Affected

Z		= Unchanged
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	= Cleared

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Write the product to the wide accumulator. Return the least-significant 64 bits of the wide accumulator in the destination register.

Any flag updates occur only if the set flags suffix (.F) is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- Using an odd destination register for DSP instructions with 64-bit 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](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
MPYD<.f>	a,b,c	00101bbb00011000FBBBCCCCC0AAAAAA
MPYD<.f>	a,b,u6	00101bbb01011000FBBBuuuuuuAAAAAA
MPYD<.f>	b,b,s12	00101bbb10011000FBBBssssssSSSSSS
MPYD<.cc><.f>	b,b,c	00101bbb11011000FBBBCCCCC0QQQQQ0
MPYD<.cc><.f>	b,b,u6	00101bbb11011000FBBBuuuuuu1QQQQQ0

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

MPYDF

Function

Signed 32x32 fractional multiplication. Return a 64-bit result.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide = b * c;
    a = SAT64(accwide<<1);
}
```

 **Note**

A is a register pair representing a 64-bit operand.

Instruction Format

op A, b, c

Syntax Example

MPYDF <.f> A,b,c

Timing Characteristics

Two cycle

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands, and write the product to the wide accumulator. Assign the saturated least 64-bits of the wide accumulator to the destination register pair.

The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- Using an odd destination register for DSP instructions with 64-bit 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	00110bbb00010010FBBBCCCCC	AAAAAA
MPYDF<.f>	a, b, u6	00110bbb01010010FBBBuuuuuu	AAAAAA
MPYDF<.f>	b, b, s12	00110bbb10010010FBBBssssss	SSSSSS
MPYDF<.cc><.f>	b, b, c	00110bbb11010010FBBBCCCCC	0QQQQQ
MPYDF<.cc><.f>	b, b, u6	00110bbb11010010FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYDU

Function

Unsigned 32x32 multiplication with a 64-bit result. For the non-DSP version, see [MPYDU](#).

Extension Group

MPY_OPTION > 7 or has_dsp==true

Operation

```
if (cc) {
    result = (b * c);
    A = result;
    acc = result;
}
```

Instruction Format

op A, b, c

Syntax Example

MPYDU <.f> A,b,c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V	•	= Cleared

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand 1 and operand 2. Write the product to the wide accumulator. Return the least-significant 64 bits of the wide accumulator in the destination register pair.

Any flag updates occur only if the set flags suffix (.F) is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- Using an odd destination register for DSP instructions with 64-bit 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



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

Instruction Code

MPYDU<.f>	a, b, c	00101 bbb 00 011001 FBBBCCCCC AAAAAA
MPYDU<.f>	a, b, u6	00101 bbb 01 011001 FBBBuuuuuu AAAAAA
MPYDU<.f>	b, b, s12	00101 bbb 10 011001 FBBBssssss SSSSSS
MPYDU<.cc><.f>	b, b, c	00101 bbb 11 011001 FBBBCCCCC 0QQQQQ
MPYDU<.cc><.f>	b, b, u6	00101 bbb 11 011001 FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYF

Function

Signed 32x32 fractional multiplication.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide = b * c;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands, and write the product to the wide accumulator. Return the truncated and saturated second 32 bits of the wide accumulator to the destination register.

The sticky flag `DSP_CTRL.SAT` is set, regardless of the `.F` suffix, if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYF*/
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -
1 : ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(accwide >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        ACC0_GHI.V      = sat_en(b * c, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -
1 : ACC0_GHI.GUARD != 0;
    else
        ACC0_GHI.V = sat(b * c, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(accwide >> 31, 31, &a);
DSP_CTRL.SAT |= s || v | ACC0_GHI.V;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a == 0;
    N_flag = a < 0;
    V_flag = v;
}
```

Assembly Code Example

```
MPYF r0,r1,r2      ; Signed fractional multiplication of r2 and r3. The
                   ;result is saturated and stored in r0.
```

Syntax and Encoding

Instruction Code		
MPYF<.f>	a,b,c	00110bbb00001010FBBBCCCCCAAAAAA
MPYF<.f>	a,b,u6	00110bbb01001010FBBBuuuuuuAAAAAA

MPYF<.f>	b,b,s12	00110 bbb 10 001010 F BBB ssssss SSSSSS
MPYF<.cc><.f>	b,b,c	00110 bbb 11 001010 F BBB CCCCC 0QQQQQ
MPYF<.cc><.f>	b,b,u6	00110 bbb 11 001010 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYFR

Function

Signed 32x32 fractional multiplication. The result is rounded.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide = b * c;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	= Set if result saturates

Description

Compute the signed fractional 32x32 product of the first and second source operands, and write the product to the wide accumulator. Return the rounded and saturated second 32 bits of the wide accumulator to the destination register.

The sticky flag DSP_CTRL.SAT is set, regardless of the .F suffix, if the result saturates. The other flags are updated only if the .F suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MPYFR*/
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -
1 : ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -
1 : ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(round(accwide,31) >> 31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = (accwide >> 32) & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a == 0;
            N_flag = a < 0;
            V_flag = s;
        }

```

Assembly Code Example

```

MPYFR r0,r1,r2    ; Signed fractional multiplication of r2 and r3. The
                  ;result is saturated, rounded, and stored in r0.

```

Syntax and Encoding

Instruction Code

MPYFR<.f>	a,b,c	00110 bbb 00 001011 F BBB CCCCC AAAAAA
MPYFR<.f>	a,b,u6	00110 bbb 01 001011 F BBB uuuuuu AAAAAA

MPYFR<.f>	b,b,s12	00110 bbb 10 001011 F BBB ssssss SSSSSS
MPYFR<.cc><.f>	b,b,c	00110 bbb 11 001011 F BBB CCCCC 0QQQQQ
MPYFR<.cc><.f>	b,b,u6	00110 bbb 11 001011 F BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYM MPYH

Function

32 x 32 Signed Multiply High For the non-DSP version, see [MPYM MPYH](#).

 **Note**

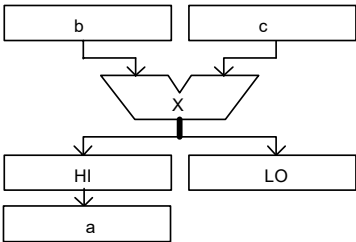
The MPYH mnemonic is deprecated.

Extension Group

- MPY_OPTION > 1 or has_dsp==true

Operation

if (cc) a = (signed) (b * c) >> 32;



Instruction Format

op a, b, c

Syntax Example

MPYM<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set is the accumulator is negative
C		= Unchanged
V	•	= Always cleared.

Description

Perform a signed 32-bit by 32-bit multiplication of operand 1 and operand 2. Return bits [63:32] of the result.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers          /* MPYM */
w  = b * c;                                     /*when has_dsp==true
a  = (w >> 32 & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = a < 0;
    V_flag = 0;
}
```

Assembly Code Example

```
MPYM r1,r2,r3      ;Multiply r2 by r3 and put high part of the result in
                   ;r1
```

Syntax and Encoding

Instruction Code

MPYM<.f>	a,b,c	00100bbb00011011FBBBCCCCCAAAAAA
MPYM<.f>	a,b,u6	00100bbb01011011FBBBuuuuuuAAAAAA
MPYM<.f>	b,b,s12	00100bbb10011011FBBBssssssSSSSSS
MPYM<.cc><.f>	b,b,c	00100bbb11011011FBBBCCCCC0QQQQQ
MPYM<.cc><.f>	b,b,u6	00100bbb11011011FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYMU MPYHU

Function

32 x 32 Unsigned integer Multiply High. For the non-DSP version, see [MPYMU](#), [MPYHU](#).



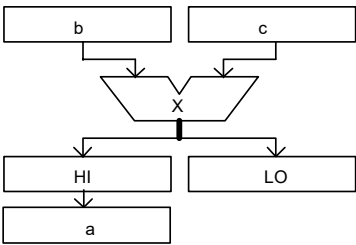
The MPYHU mnemonic is deprecated.

Extension Group

- MPY_OPTION > 1 or has_dsp==true

Operation

if (cc) a = (unsigned) (b * c) >> 32;dest (src1 X src2).high



Instruction Format

op a, b, c

Syntax Example

MPYMU<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared.
C		= Unchanged
V	•	= Always cleared.

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand 1 and operand 2. Return bits [63:32] of the result.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers          /* MPYMU */
t = (unsigned) b * c;                          /* when has_dsp==true */
a = (t >> 32 & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = 0;
    V_flag = 0;
}
```

Assembly Code Example

```
MPYMU r1,r2,r3    ;Multiply r2 by r3 and put high part of the result
                  ;in r1
```

Syntax and Encoding

Instruction Code

MPYMU<.f>	a,b,c	00100bbb00011100FBBBCCCCCAAAAAA
MPYMU<.f>	a,b,u6	00100bbb01011100FBBBuuuuuuAAAAAA
MPYMU<.f>	b,b,s12	00100bbb10011100FBBBssssssSSSSSS
MPYMU<.cc><.f>	b,b,c	00100bbb11011100FBBBCCCCC0QQQQQ
MPYMU<.cc><.f>	b,b,u6	00100bbb11011100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYU

Function

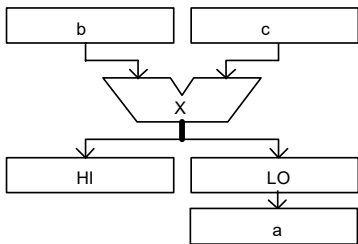
32 x 32 Unsigned Multiply, returning least-significant 32-bit word of result. For the non-DSP version, see [MPYU](#).

Extension Group

(MPY_OPTION > 1) or (has_dsp == true)

Operation

if (cc) a = (unsigned) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPYU<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Set when the result cannot be represented as a 32-bit unsigned integer

Description

Perform an unsigned 32-bit by 32-bit multiplication of operand 1 and operand 2. Return the least-significant bits of the product to the destination register.

Any flag updates occur only if the set flags suffix (.F) is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers          /* 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	00100 bbb 00 011101 FBBB CCCCC AAAAAA
MPYU<.f>	a,b,u6	00100 bbb 01 011101 FBBB uuuuuu AAAAAA
MPYU<.f>	b,b,s12	00100 bbb 10 011101 FBBB ssssss SSSSSS
MPYU<.cc><.f>	b,b,c	00100 bbb 11 011101 FBBB CCCCC 0QQQQQ
MPYU<.cc><.f>	b,b,u6	00100 bbb 11 011101 FBBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYUW

Function

16 x 16 unsigned Multiply, return 32-bit word result. For the non-DSP version, see [MPYUW](#), [MPYUW_S](#).

Extension Group

(MPY_OPTION > 0) or (has_dsp == true)

Operation

For has_dsp==true

```
if (cc) {
    accwide = (unsigned) b.h0 * c.h0;
    a = accwide.w0;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYUW<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Always cleared

Description

Perform an unsigned 16-bit by 16-bit multiply of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register. The 16-bit source operands use the lower 16 bits of the source registers. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
// note: data modeled as 128b integers
t = (unsigned)b.h0 * c.h0;
a = (t & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = 0;
    V_flag = 0;
}
```

```
/* MPYUW */
/* when has_dsp==true
```

Assembly Code Example

```
MPYUW r1,r2, r3 ; Multiply r2 by r3 and put the result in r1
```

Syntax and Encoding

Instruction Code

MPYUW<.f>	a,b,c	00100bbb00011111FBBBCCCCCAAAAAA
MPYUW<.f>	a,b,u6	00100bbb01011111FBBBuuuuuuAAAAAA
MPYUW<.f>	b,b,s12	00100bbb10011111FBBBssssssSSSSSS
MPYUW<.cc><.f>	b,b,c	00100bbb11011111FBBBCCCCC0QQQQQ
MPYUW<.cc><.f>	b,b,u6	00100bbb11011111FBBBuuuuuu1QQQQQ
MPYUW_S	b,b,c	01111bbbccc01010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYUW_S

Function

Unsigned 16 x 16 multiplication, return a 32-bit result. For the non-DSP version, see [MPYUW](#), [MPYUW_S](#).

Extension Group

(MPY_OPTION > 0) or (has_dsp == true)

Operation

```
if (cc) {
    t = (unsigned) b.h0 * c.h0;
    b = t.w0;
}
```

Instruction Format

op b, c

Syntax Example

MPYUW_S<.f> b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an unsigned 16-bit by 16-bit multiply of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register. The 16-bit source operands use the lower 16 bits of the source registers. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers                                /* MPYUW_S */
t = (unsigned)b.h0 * c.h0;
b = (t & 0xffff_ffff);
```


Assembly Code Example

```
MPYUW_S r1, r2    ; Unsigned 16x16 multiplication of r2 and r1 and
                  ;return the result in r1
```

Syntax and Encoding

Instruction Code

MPYUW_S	b,c	01111bbbccc01010

MPYW

Function

16 x 16 Signed Multiply. For the non-DSP version, see [MPYW](#), [MPYW_S](#).

Extension Group

(MPY_OPTION > 0) or (has_dsp == true)

Operation

For has_dsp==true

```
if (cc) {
    accwide = b.h0 * c.h0;
    a = accwide.w0;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYW<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 32-bit result is set
C		= Unchanged
V	•	= Set when the unsigned result overflows

Description

Perform a signed 16-bit by 16-bit multiply of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register. The 16-bit source operands use the lower 16 bits of the source registers. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
// note: accumulators modeled as 128b integers      /* MPYW */
t  = b.h0 * c.h0;                                   /* when has_dsp==true
a  = (t & 0xffff_ffff);
if F {
    Z_flag = a == 0;
    N_flag = t < 0;
    V_flag = (t >> 32) != 0;
}
```

Assembly Code Example

```
MPYW r1,r2, r3    ; Multiply r2 by r3 and put the result in r1
```

Syntax and Encoding

Instruction Code

MPYW<.f>	a,b,c	00100bbb00011110FBBBCCCCCAAAAAA
MPYW<.f>	a,b,u6	00100bbb01011110FBBBuuuuuuAAAAAA
MPYW<.f>	b,b,s12	00100bbb10011110FBBBssssssSSSSSS
MPYW<.cc><.f>	b,b,c	00100bbb11011110FBBBCCCCC0QQQQQ
MPYW<.cc><.f>	b,b,u6	00100bbb11011110FBBBuuuuuu1QQQQQ
MPYW_S	b,b,c	01111bbbccc01001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYW_S

Function

Signed 16 x 16 multiplication, whole 32-bit word result. For the non-DSP version, see [MPYW](#), [MPYW_S](#).

Extension Group

(MPY_OPTION > 0) or (has_dsp == true)

Operation

```
t = b.h0 * c.h0;
b = t.w0;
```

Instruction Format

op b, c

Syntax Example

MPYW_S<.f> b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed 16-bit by 16-bit multiply of operand 1 and operand 2. Return the least-significant 32 bits of the result in the destination register. The 16-bit source operands use the lower 16 bits of the source registers. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: data modeled as 128b integers                                /* MPYW_S */
t  = b.h0 * c.h0;
b  = (t & 0xffff_ffff);
```

Assembly Code Example

```
MPYW_S r1, r2      ; Signed 16x16 multiplication of r2 and r1 and
                    ;return the result in r1
```

Syntax and Encoding

Instruction Code		
MPYW_S	b,c	01111bbbccc01001

MPYWHFL

Function

Signed fractional multiplication of the 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h0 <<16;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHFL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of the 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the saturated most-significant 32 bits of the wide accumulator without the guard bits to the destination operand. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c.h0 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h0 << 16, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(accwide>>31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
/* MPYWHFL
*/
```

Assembly Code Example

```
MPYWHFL r0,r2,r3      ; Signed fractional low multiply r2 and r3 and
                      ; return result in register r0
```

Syntax and Encoding

Instruction Code		
MPYWHFL<.f>	a,b,c	00110bbb00100100FBBBCCCCCAAAAAA
MPYWHFL<.f>	a,b,u6	00110bbb01100100FBBBuuuuuuAAAAAA

MPYWHFL<.f>	b, b, s12	00110bbb10100100FBBBssssssSSSSSS
MPYWHFL<.cc><.f>	b, b, c	00110bbb11100100FBBBCCCCC0QQQQQ
MPYWHFL<.cc><.f>	b, b, u6	00110bbb11100100FBBBuuuuuu100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHFLR

Function

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. The result is saturated and rounded.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h0 <<16;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHFLR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the saturated and rounded most-significant 32 bits of the wide accumulator without the guard bits to the destination operand. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYWHFLR
s = 0;                                                                    */
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c.h0 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h0 << 16, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(round(accwide, 31) >> 31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
```

Assembly Code Example

```
MPYWHFLR r0,r2,r3      ; Signed fractional low multiply r2 and r3 and
                        ; return the saturated and rounded result in
                        ;register r0
```

Syntax and Encoding

Instruction Code

```
MPYWHFLR<.f>          a,b,c          00110bbb00100101FBBBCCCCCAAAAA
```

MPYWHFLR<.f>	a,b,u6	00110bbb01100101FBBBuuuuuuAAAAAA
MPYWHFLR<.f>	b,b,s12	00110bbb10100101FBBBssssssSSSSSS
MPYWHFLR<.cc><.f>	b,b,c	00110bbb11100101FBBBCCCCC000000
MPYWHFLR<.cc><.f>	b,b,u6	00110bbb11100101FBBBuuuuuu100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHFM

Function

Signed fractional multiplication of 32 bits of one operand and most-significant 16 bits of the other operand.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h1 <<16;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHFM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the saturated most-significant 32 bits of the wide accumulator without the guard bits to the destination operand. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        ACC0_GHI.V      = sat_en(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V      = sat_en(b * c.h1 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G      = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h1 << 16, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(accwide>>31, 31, &a);
        DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
        ACC0_HI = accwide >> 32 & 0xffff_ffff;
        ACC0_LO = accwide & 0xffff_ffff;
        ACC0_GHI.Z = accwide == 0;
        ACC0_GHI.N = accwide < 0;
        if F {
            Z_flag = a = 0;
            N_flag = a < 0;
            V_flag = v;
        }
/* MPYWHFM
*/
```

Assembly Code Example

```
MPYWHFM r0,r2,r3      ; Signed fractional high multiply r2 and r3 and
                      ; return result in register r0
```

Syntax and Encoding

Instruction Code

MPYWHFM<.f>	a,b,c	00110bbb00100000FBBBCCCCCAAAAAA
MPYWHFM<.f>	a,b,u6	00110bbb01100000FBBBuuuuuuAAAAAA

MPYWHFM<.f>	b, b, s12	00110bbb10100000FBBBssssssSSSSSS
MPYWHFM<.cc><.f>	b, b, c	00110bbb11100000FBBBCCCCC0QQQQQ
MPYWHFM<.cc><.f>	b, b, u6	00110bbb11100000FBBBuuuuuu100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHFMR

Function

Signed fractional multiplication of 32 bits of one operand and most-significant 16 bits of the other operand. The result is rounded and saturated.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h1 <<16;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHFMR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the saturated most-significant 32 bits of the wide accumulator without the guard bits to the destination operand. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        ACC0_GHI.V = sat(add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        ACC0_GHI.V = sat(add, 63, &accwide);
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            ACC0_GHI.V = sat(b * c.h1 << 16, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            ACC0_GHI.V = sat(b * c.h1 << 16, 63, &accwide);
            v = sat(round(accwide, 31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}
```

Assembly Code Example

```
MPYWHFMR r0,r2,r3      ; Signed fractional high multiply r2 and r3 and
                        ; return result in register r0
```

Syntax and Encoding

Instruction Code

MPYWHFMR<.f>	a,b,c	00110bbb00100001FBBBCCCCCAAAAAA
MPYWHFMR<.f>	a,b,u6	00110bbb01100001FBBBuuuuuuAAAAAA
MPYWHFMR<.f>	b,b,s12	00110bbb10100001FBBBssssssSSSSSS

MPYWHFMR<.cc><.f>	b, b, c	00110bbb11100001FBBBCCCCC0QQQQQ
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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHL

Function

Signed multiplication of 32 bits of one operand and least-significant 16 bits of the other operand.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = b * c.h0;
    a = accwide;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	=Set if the accumulator cannot be represented as a 32-bit signed integer

Description

Signed multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the least-significant 32 bits of the wide accumulator. The other flags are updated only if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide = b * c.h0;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
    s = accwide >> 31 != 0 && accwide >> 31 != -1
else
    accwide = b * c.h0;
    s = accwide >> 31 != 0 && accwide >> 31 != -1
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    Z_flag = a == 0;
    N_flag = accwide < 0;
    V_flag |= s;
}
```

Assembly Code Example

```
MPYWHL r0,r2,r3      ; Signed integer multiplication r2 and r3 and
                     ; return result in register r0
```

Syntax and Encoding

Instruction Code		
MPYWHL<.f>	a, b, c	00110bbb000111100FBBBCCCCC AAAAAA
MPYWHL<.f>	a, b, u6	00110bbb010111100FBBBuuuuuu AAAAAA
MPYWHL<.f>	b, b, s12	00110bbb100111100FBBBssssss SSSSSS
MPYWHL<.cc><.f>	b, b, c	00110bbb110111100FBBBCCCCC 0QQQQQ
MPYWHL<.cc><.f>	b, b, u6	00110bbb110111100FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

MPYWHUL

Function

Unsigned multiplication of 32 bits of one operand and the least-significant 16 bits of the other operand.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide = (unsigned) (b * c.h0);
    a = accwide;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHUL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register value is zero
N	•	= Cleared
C		= Unchanged
V	•	=Set if the accumulator cannot be represented as a 32-bit unsigned integer

Description

Unsigned multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Write the product to the wide accumulator. Return the least-significant 32 bits of the wide accumulator. The other flags are updated only if the *.F* suffix is used. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
if (DSP_CTRL.GE)
    accwide = b * c.h0;
    ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
    s = accwide >> 32 != 0;
else
    accwide = b * c.h0;
    s = accwide >> 32 != 0;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
a = accwide & 0xffff_ffff;
if F {
    Z_flag = a == 0;
    N_flag = a < 0;
    V_flag |= s;
}
```

/* MPYWHUL
*/

Assembly Code Example

```
MPYWHUL r0,r2,r3      ; Unsigned integer multiplication r2 and r3 and
                      ; return result in register r0
```

Syntax and Encoding

Instruction Code		
MPYWHUL<.f>	a, b, c	00110bbb000111110FBBBCCCCC AAAAAA
MPYWHUL<.f>	a, b, u6	00110bbb010111110FBBBuuuuuu AAAAAA
MPYWHUL<.f>	b, b, s12	00110bbb100111110FBBBssssss SSSSSS
MPYWHUL<.cc><.f>	b, b, c	00110bbb110111110FBBBCCCCC 0QQQQQ
MPYWHUL<.cc><.f>	b, b, u6	00110bbb110111110FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

MPYWHKL

Function

Signed integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    acchi = b * c.h0 >> 16;
    a = acchi;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHKL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the accumulator is negative
C		= Unchanged
V	•	=Set if the accumulator cannot be represented by a 32-bit signed integer

Description

Signed integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator. Return the least-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYWHKL */
if (DSP_CTRL.GE)
    acchi = (b * c.h0) >> 16;
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
else
    acchi = (b * c.h0) >> 16;
    ACC0_HI = acchi & 0xffff_ffff;
    a = acchi & 0xffff_ffff;
    if (DSP_CTRL.GE)
        rem = acchi >> 39;
    else
        rem = acchi >> 31;
    if F {
        N_flag = rem & 1;
        V_flag = rem != -1 && rem != 0;
        Z_flag = a == 0;
    }
```

Assembly Code Example

```
MPYWHKL r0,r2,r3                ; Signed integer multiplication of r2 and r3
                                ; shift result by 16 bits, and return the shifted
                                ; result in register r0
```

Syntax and Encoding

Instruction Code

MPYWHKL<.f>	a, b, c	00110bbb00101010FBBBCCCCC AAAAAA
MPYWHKL<.f>	a, b, u6	00110bbb01101010FBBBuuuuuu AAAAAA
MPYWHKL<.f>	b, b, s12	00110bbb10101010FBBBssssss SSSSSS
MPYWHKL<.cc><.f>	b, b, c	00110bbb11101010FBBBCCCCC 0QQQQQ
MPYWHKL<.cc><.f>	b, b, u6	00110bbb11101010FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MPYWHKUL

Function

Unsigned integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    acchi = b * c.h0 >> 16;
    a = acchi;
}
```

Instruction Format

op a, b, c

Syntax Example

MPYWHKUL <.f> a,b,c

STATUS32 Flags Affected

Z		= Set if the result is zero
N	•	= Cleared
C		= Unchanged
V	•	=Set if the accumulator cannot be represented as a 32-bit unsigned integer

Description

Unsigned integer multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Right-shift the product by 16 bits and write the shifted product to the high accumulator. Return the least-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                /* MPYWHKUL */
if (DSP_CTRL.GE)
    acchi = (b * c.h0) >> 16;
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    s = acchi >> 40 != 0;
else
    acchi = (b * c.h0;) >> 16
    s = acchi >> 32 != 0;
ACC0_HI = acchi & 0xffff_ffff;
a = acchi & 0xffff_ffff;
if F {
    V_flag = s;
    N_flag = 0;
    Z_flag = a == 0;
}
```

Assembly Code Example

```
MPYWHKUL r0,r2,r3      ; Unsigned integer multiplication of r2 and r3
                       ; shift result by 16 bits, and return the shifted
                       ; result in register r0
```

Syntax and Encoding

Instruction Code

MPYWHKUL<.f>	a,b,c	00110bbb00101011FBBBCCCCC	AAAAAA
MPYWHKUL<.f>	a,b,u6	00110bbb01101011FBBBuuuuuu	AAAAAA
MPYWHKUL<.f>	b,b,s12	00110bbb10101011FBBBssssss	SSSSSS
MPYWHKUL<.cc><.f>	b,b,c	00110bbb11101011FBBBCCCCC	0QQQQQ
MPYWHKUL<.cc><.f>	b,b,u6	00110bbb11101011FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

MSUBDF

Function

Signed 32x32 fractional multiplication and subtraction; return a 64-bit result.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide -= b * c;
    A = SAT64(accwide<<1);
}
```

Instruction Format

op A, b, c

Syntax Example

MSUBDF <.f> A,b,c

Timing Characteristics

Two cycle

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Perform a signed fractional multiplication of 32 bits of operand *b* and operand *c*; subtract the product from the wide accumulator. Return the saturated least-significant 64 bits of the wide accumulator.

The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (`LP_COUNT/r60`) as a destination operand register.
- Using an odd destination register for DSP instructions with 64-bit 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	00110bbb00010101FBBBCCCCC AAAAAA
MSUBDF<.f>	a, b, u6	00110bbb01010101FBBBuuuuuu AAAAAA
MSUBDF<.f>	b, b, s12	00110bbb10010101FBBBssssss SSSSSS
MSUBDF<.cc><.f>	b, b, c	00110bbb11010101FBBBCCCCC 0QQQQQ
MSUBDF<.cc><.f>	b, b, u6	00110bbb11010101FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBF

Function

Signed 32x32 fractional multiplication and subtraction; return a 32-bit result.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide -= b * c;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBF <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Perform a signed fractional multiplication of the 32 bits of the operands *b* and operand *c*; subtract the product from the wide accumulator. Return the saturated second 32-bit word of the wide accumulator. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBF */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
            ACC0_GHI.V = sat_en(accwide - b * c, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
            ACC0_GHI.V = sat(accwide - b * c, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(accwide >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MSUBF r1,r2,r3          ; Signed fractional multiplication of r2 and r3
                        ;and subtract the product from the accumulator
                        ; return result in register r1

```

Syntax and Encoding

		Instruction Code
MSUBF<.f>	a, b, c	00110bbb00001110FBBBCCCCCAAAAA
MSUBF<.f>	a, b, u6	00110bbb01001110FBBBuuuuuuAAAAA
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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBFR

Function

Signed 32x32 fractional multiplication and subtraction; return a rounded and saturated 32-bit result.

Extension Group

has_dsp==true

Operation

```
if (cc) {
    accwide -= b * c;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBFR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Perform a signed fractional multiplication of the 32 bits of the operands *b* and operand *c*; subtract the product from the wide accumulator. Return the rounded and saturated second 32-bit word of the wide accumulator. The sticky flag `DSP_CTRL.SAT` is set regardless of the `.F` suffix if the result saturates. The other flags are updated only if the `.F` suffix is used. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c << 1;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c << 1), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
    else
        if (DSP_CTRL.GE)
            accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO)
            ACC0_GHI.V = sat_en(accwide - b * c, 71, &accwide);
            ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
            ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO)
            ACC0_GHI.V = sat(accwide - b * c, 63, &accwide);
            ACC0_GHI.G = 0;
            v = sat(round(accwide,31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}
```

Assembly Code Example

```
MSUBFR r1,r2,r3      ; Signed fractional multiplication of r2 and r3 and
                     ;subtract the product from the accumulator
                     ; return rounded and saturated result in register r1
```

Syntax and Encoding

Instruction Code		
MSUBFR<.f>	a, b, c	00110bbb00001111FBBBCCCCCAAAAA
MSUBFR<.f>	a, b, u6	00110bbb01001111FBBBuuuuuuAAAAA
MSUBFR<.f>	b, b, s12	00110bbb10001111FBBBssssssSSSSSS
MSUBFR<.cc><.f>	b, b, c	00110bbb11001111FBBBCCCCC0QQQQQ
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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBWHFL

Function

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide -= b * c.h0 <<32;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBWHFL <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBWHFL */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - b * c.h0 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.G = 0;
        ACC0_GHI.V = sat(accwide - b * c.h0 << 16, 63, &accwide); v =
sat(accwide>>31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MSUBWHFL r0,r2,r3          ; Signed fractional multiply-subtract r2 and r3
                           ;and return result in register r0

```


Syntax and Encoding

		Instruction Code
MSUBWHFL<.f>	a, b, c	00110bbb00010100FBBBCCCCCAAAAA
MSUBWHFL<.f>	a, b, u6	00110bbb01010100FBBBuuuuuuAAAAA
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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBWHFLR

Function

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the rounded and saturated result.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide -= b * c.h0 << 32;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBWHFLR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the least-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the rounded and saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* MSUBWHFLR
s = 0;                                                                           */
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h0 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h0 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide,32) >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - b * c.h0 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - b * c.h0 << 16, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(round(accwide, 31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MSUBWHFLR r0,r2,r3      ; Signed fractional multiply-subtract r2 and r3
                        ;and return result in register r0

```

Syntax and Encoding

		Instruction Code
MSUBWHFLR<.f>	a, b, c	00110bbb00011010FBBBCCCCC AAAAAA
MSUBWHFLR<.f>	a, b, u6	00110bbb01011010FBBBuuuuuu AAAAAA
MSUBWHFLR<.f>	b, b, s12	00110bbb10011010FBBBssssss SSSSSS
MSUBWHFLR<.cc><.f>	b, b, c	00110bbb11011010FBBBCCCCC 0QQQQQ
MSUBWHFLR<.cc><.f>	b, b, u6	00110bbb11011010FBBBuuuuuu 100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBWHFM

Function

Signed fractional multiplication and subtraction of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand.

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide -= b * c.h1 <<32;
    a = SAT32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBWHFM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBWHFM */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
        v = sat(accwide>>32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - b * c.h1 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - b * c.h1 << 16, 63, &accwide); v =
sat(accwide>>31, 31, &a);
        ACC0_GHI.G = 0;
    DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
    ACC0_HI = accwide >> 32 & 0xffff_ffff;
    ACC0_LO = accwide & 0xffff_ffff;
    ACC0_GHI.Z = accwide == 0;
    ACC0_GHI.N = accwide < 0;
    if F {
        Z_flag = a = 0;
        N_flag = a < 0;
        V_flag = v;
    }

```

Assembly Code Example

```

MSUBWHFM r0,r2,r3          ; Signed fractional multiply-subtract r2 and r3 and
                           ; return the rounded and saturated result in register r0

```

Syntax and Encoding

		Instruction Code
MSUBWHFM<.f>	a, b, c	00110 bbb 00101100FBBBCCCCC AAAAAA
MSUBWHFM<.f>	a, b, u6	00110 bbb 01101100FBBBuuuuuu AAAAAA
MSUBWHFM<.f>	b, b, s12	00110 bbb 10101100FBBBssssss SSSSSS
MSUBWHFM<.cc><.f>	b, b, c	00110 bbb 11101100FBBBCCCCC 0QQQQQ
MSUBWHFM<.cc><.f>	b, b, u6	00110 bbb 11101100FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

MSUBWHFMR

Function

Signed fractional multiplication and subtraction of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand; with rounded result

Extension Group

DSP 32x16 MAC

Operation

```
if (cc) {
    accwide -= b * c.h1 << 32;
    a = RND32(accwide>>31);
}
```

Instruction Format

op a, b, c

Syntax Example

MSUBWHFMR <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set if the result is negative
C		= Unchanged
V	•	=Set if the result saturates

Description

Signed fractional multiplication of 32 bits of the *b* operand and the most-significant 16 bits of the *c* operand. Subtract the product from the wide accumulator. Return the rounded and saturated most-significant 32 bits of the accumulator without the guard bits. Flags are updated only if the *.F* suffix is used. The `DSP_CTRL.SAT` flag is set regardless of the *.F* suffix. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* MSUBWHFMR */
s = 0;
if (DSP_CTRL.PA)
    if (DSP_CTRL.GE)
        add = b * c.h1 << 17;
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - add, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        s = sat((b * c.h1 << 17), 63, &add);
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - add, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(round(accwide,32) >> 32, 31, &a);
else
    if (DSP_CTRL.GE)
        accwide = sext(ACC0_GHI.GUARD)<<64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
        ACC0_GHI.V = sat_en(accwide - b * c.h1 << 16, 71, &accwide);
        ACC0_GHI.GUARD = (accwide >> 64) & 0xff;
        ACC0_GHI.G = ((accwide >> 63) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    else
        accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
        ACC0_GHI.V = sat(accwide - b * c.h1 << 16, 63, &accwide);
        ACC0_GHI.G = 0;
    v = sat(round(accwide, 31) >> 31, 31, &a);
DSP_CTRL.SAT |= s || v || ACC0_GHI.V;
ACC0_HI = accwide >> 32 & 0xffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_GHI.Z = accwide == 0;
ACC0_GHI.N = accwide < 0;
if F {
    Z_flag = a = 0;
    N_flag = a < 0;
    V_flag = v;
}

```

Assembly Code Example

```

MSUBWHFMR r0,r2,r3      ; Signed fractional multiply-subtract r2 and r3 and
                        ; return the rounded and saturated result in register r0

```

Syntax and Encoding

		Instruction Code
MSUBWHFMR<.f>	a, b, c	00110 bbb 00101101FBBBCCCCC AAAAAA
MSUBWHFMR<.f>	a, b, u6	00110 bbb 01101101FBBBuuuuuu AAAAAA
MSUBWHFMR<.f>	b, b, s12	00110 bbb 10101101FBBBssssss SSSSSS
MSUBWHFMR<.cc><.f>	b, b, c	00110 bbb 11101101FBBBCCCCC 0QQQQQ
MSUBWHFMR<.cc><.f>	b, b, u6	00110 bbb 11101101FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

NEGS

Function

Negate a 32-bit word and saturate the result.

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT32}(-c);$$

Instruction Format

op b,c

Syntax Example

NEGS<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if input is 0x8000_0000; otherwise cleared

Description

Negate long word operand, c, and place the result in the destination register b while saturating. An absolute source value of 0x8000_0000 yields 0x7FFF_FFFF. The saturation flag, DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. If the set flags suffix (.F) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = (c == 0x8000_0000 ? 0x7fff_ffff : -c);          /* NEGS */
DSP_CTRL.SAT |= c == 0x8000_0000;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = b < 0;
    STATUS32.V = c == 0x8000_0000;
}
```

Assembly Code Example

```
NEGS r1,r2    ; Negate r2 and write saturated result into r1
```

Syntax and Encoding

Instruction Code

NEGS<.f>	b,c	00101 bbb 00 000111 F BBB CCCCC 000111
NEGS<.f>	b,limm	00101 bbb 00 101111 F BBB 111110 000111
NEGS<.f>	0,c	00101 110 00 101111 F 111 CCCCC 000111
NEGS<.f>	0,limm	00101 110 00 101111 F 111 111110 000111
NEGS<.f>	b,u6	00101 bbb 01 000111 F BBB uuuuuu 000111
NEGS<.f>	0,u6	00101 110 01 101111 F 111 uuuuuu 000111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

NEGSH

Function

Negate the lower 16 bits of a word and saturate the result.

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT16}(-c);$$

Instruction Format

op b,c

Syntax Example

NEGSH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if input is 0x8000_0000; otherwise cleared

Description

Negate lower 16 bits of the operand, c, and place the result in the destination register b while saturating. An absolute source value of 0x8000_0000 yields 0x7FFF_FFFF. The saturation flag, DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. If the set flags suffix (.F) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
t = c & 0x0000_ffff;                                /* NEGSH */
b = (t == 0x0000_8000 ? 0x00007fff : -t);
DSP_CTRL.SAT |= t == 0x0000_8000;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = b < 0;
    STATUS32.V = t == 0x0000_8000;
}
```

Assembly Code Example

```
NEGSH r1,r2    ; Negate lower 16 bits of r2 and write saturated result
               ;into r1
```

Syntax and Encoding

Instruction Code

NEGSH<.f>	b,c	00101 bbb 00 101111 FBBB CCCCC 000110
NEGSH<.f>	b,limm	00101 bbb 00 101111 FBBB 111110 000110
NEGSH<.f>	0,c	00101 110 00 101111 F111 CCCCC 000110
NEGSH<.f>	0,limm	00101 110 00 101111 F111 111110 000110
NEGSH<.f>	0,u6	00101 110 01 101111 F111 uuuuuu 000110
NEGSH<.f>	b,u6	00101 bbb 01 101111 FBBB uuuuuu 000110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

NORMACC

Function

Normalize the accumulator.

Extension Group

DSP Accumulator

Operation

```
switch (c.b1) {  
0: b.b0 = norm(accwide);  
1: b.b0 = norm(acclo);  
2: b.b0 = norm(acchi);  
3: b.b2 = norm(acchi); b.b0 = norm(acclo);  
}
```

Instruction Format

op b,c

Syntax Example

NORMACC b, c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Return the left shift amount for normalizing an accumulator to the fractional dot. C operand consists of one field: the accumulator select. Return the result in B. B is a signed byte and is in the range $-8 \leq B \leq 64$. The result is a vector in case the vector accumulator is selected.

Pseudo Code

```

short norm_var(int128 c, int w) {                                     /* NORMACC */
    int i;
    for (i = 0; (i < w) && ((c >> i) != 0) && ((c >> i) != -1); i++);
    return w - i - 1;
}
switch (c.b1) {
    case 0:
        if (DSP_CTRL.GE)
            accwide = ACC0_GHI.GUARD << 64 | unsigned(ACC0_HI) << 32 |
unsigned(ACC0_LO);
            if (DSP_CTRL.PA)
                B = norm_var(accwide, 72)-8;
            else
                B = norm_var(accwide, 72)-9;
        else
            accwide = ACC0_HI << 32 | unsigned(ACC0_LO);
            if (DSP_CTRL.PA)
                B = norm_var(accwide, 64);
            else
                B = norm_var(accwide, 64)-1;
        B &= 0xff;
        break;
    case 1:
        if (DSP_CTRL.GE)
            acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
            if (DSP_CTRL.PA)
                B = norm_var(acclo, 40)-8;
            else
                B = norm_var(acclo, 40)-9;
        else
            acclo = ACC0_LO;
            if (DSP_CTRL.PA)
                B = norm_var(acclo, 32);
            else
                B = norm_var(acclo, 32)-1;
        B &= 0xff;
        break;
    case 2:
        if (DSP_CTRL.GE)
            acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
            if (DSP_CTRL.PA)
                B = norm_var(acchi, 40)-8;
            else
                B = norm_var(acchi, 40)-9;
        else
            acchi = ACC0_HI;
            if (DSP_CTRL.PA)
                B = norm_var(acchi, 32);
            else
                B = norm_var(acchi, 32)-1;
        B &= 0xff;
        break;
}

```



```
case 3:
    if (DSP_CTRL.GE)
        acclo = ACC0_GLO.GUARD << 32 | unsigned(ACC0_LO);
        if (DSP_CTRL.PA)
            BL = norm_var(acclo, 40)-8;
        else
            BL = norm_var(acclo, 40)-9;
    else
        acclo = ACC0_LO;
        if (DSP_CTRL.PA)
            BL = norm_var(acclo, 32);
        else
            BL = norm_var(acclo, 32)-1;
    if (DSP_CTRL.GE)
        acchi = ACC0_GHI.GUARD << 32 | unsigned(ACC0_HI);
        if (DSP_CTRL.PA)
            BH = norm_var(acchi, 40)-8;
        else
            BH = norm_var(acchi, 40)-9;
    else
        acchi = ACC0_HI;
        if (DSP_CTRL.PA)
            BH = norm_var(acchi, 32);
        else
            BH = norm_var(acchi, 32)-1;
    B = (BH && 0x000000ff) << 16 | (BL & 0x000000ff);
    break;
}
```

Assembly Code Example

```
NORMACC r1,r2      ; Normalize the accumulator value based on fields in r2;
                   ;return result to r0
```

Syntax and Encoding

Instruction Code		
NORMACC	b,c	00101bbb001011110BBBCCCCC011001
NORMACC	b,limm	00101bbb001011110BBB111110011001
NORMACC	0,c	00101110001011110111CCCCC011001
NORMACC	0,limm	00101110001011110111111110011001
NORMACC	b,u6	00101bbb011011110BBBuuuuuu011001
NORMACC	0,u6	00101110011011110111uuuuuu011001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

QMACH

Function

Quad 16x16 multiply and accumulate. For the non-DSP version, see [QMACH](#).

Extension Group

(MPY_OPTION == 9) or (dsp_wide==true)

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"](#).

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 the 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_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

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
QMACH<.f>	a,b,c	00101bbb00110100FBBBCCCCCAAAAAA
QMACH<.f>	a,b,u6	00101bbb01110100FBBBuuuuuuAAAAAA
QMACH<.f>	b,b,s12	00101bbb10110100FBBBsssssssSSSSSS
QMACH<.cc><.f>	b,b,c	00101bbb11110100FBBBCCCCC0QQQQQ
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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

QMACHF

Function

Quad 16x16 signed fractional multiply and accumulate; result is saturated

Extension Group

dsp_wide==true

Operation

```
if (cc) {
    acchi += b.h3*c.h3+b.h2*c.h2 + b.h1*c.h1+b.h0*c.h0
    a.h1 = SAT16(acchi <<1);
}
```



Note

$h0, h1, h2$, and $h3$ represent the 16-bit elements of a 64-bit operand. 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”](#).

Instruction Format

op a,B,C

Syntax Example

QMACHF <.f> a,B,C

STATUS32 Flags Affected

Z	•	= Set if the result is 0
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	= Set if the an overflow occurs during accumulation.

Description

The B and C operands specify 64-bit values, each containing four 16-bit fractional elements. Each pair of fractional elements in B and C are multiplied to form four fractional products. The four products are added to the accumulator to form the result. Return the sum of the products with the accumulator as a 16-bit saturated fraction to the destination register.

Flags are updated only if the set flags suffix (.F) is used.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /*QMACHF*
s = 0;                                                                           /
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        add0 = b.h0 * c.h0 << 1;
        add1 = b.h1 * c.h1 << 1;
        add2 = b.h2 * c.h2 << 1;
        add3 = b.h3 * c.h3 << 1;
        acchi      = acc_add(acchi, add0+add1+add2+add3, 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, &add0);
        s |= sat((b.h1 * c.h1 << 1), 31, &add1);
        s |= sat((b.h2 * c.h2 << 1), 31, &add2);
        s |= sat((b.h3 * c.h3 << 1), 31, &add3);
        acchi      = acc_add(acchi, add0+add1+add2+add3, 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
        add0 = b.h0 * c.h0;
        add1 = b.h1 * c.h1;
        add2 = b.h2 * c.h2;
        add3 = b.h3 * c.h3;
        if (DSP_CTRL.GE)
            // guard bits are enabled
            acchi      = acc_add(acchi, add0+add1+add2+add3, 40,
                                   &ACC0_GHI.Z, &ACC0_GHI.N, &ACC0_GHI.V)
            ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
            ACC0_GHI.G     = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 && ACC0_GHI.GUARD != 0;
            ACC0_HI        = acchi & 0xffff_ffff;
            v              = sat(acchi<<1, 31, &ahi);
        else
            // guard bits are disabled
            acchi      = acc_add(acchi, add0+add1+add2+add3, 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

```
QMACHF r0,r2,r4    ; Signed fractional multiply-accumulate quad vectors
                   ; {r2,r3} and {r4,r5} and return result in r0
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

QMACHF<.f>	a,b,c	00101bbb00111110FBBBCCCCCAAAAA
QMACHF<.f>	a,b,u6	00101bbb01111110FBBBuuuuuuAAAAA
QMACHF<.f>	b,b,s12	00101bbb10111110FBBBssssssSSSSSS
QMACHF<.cc><.f>	b,b,c	00101bbb11111110FBBBCCCCC0QQQQQ
QMACHF<.cc><.f>	b,b,u6	00101bbb11111110FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

QMACHU

Function

Quad unsigned 16x16 multiply and accumulate. For the non-DSP version, see [QMACHU](#).

Extension Group

(MPY_OPTION == 9) or (dsp_wide==true)

Operation

```
accwide += b.h3*c.h3+b.h2*c.h2 + b.h1*c.h1+b.h0*c.h0
a = accwide;
```



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"](#).

Instruction Format

op A,B,C

Syntax Example

QMACHU <.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

The `B` and `C` operands specify 64-bit values, each containing four 16-bit elements. Each pair of four elements in `B` and `C` is 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.

Pseudo Code

```
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.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_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
if (DSP_CTRL.GE)
    rem = accwide >> 72;
else
    rem = accwide >> 64;
if F {
    V_flag |= rem != 0;
}
```

Assembly Code Example

```
QMACHU r0,r2,r4    ; quad 16x16 unsigned multiplication and
                   ;accumulation of (r3,r2) and (r5,r4). The result is
                   ;stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
QMACHU<.f>	a,b,c	00101bbb00110101FBBBCCCCCAAAAA
QMACHU<.f>	a,b,u6	00101bbb01110101FBBBuuuuuuAAAAAA
QMACHU<.f>	b,b,s12	00101bbb10110101FBBBsssssssSSSSSS
QMACHU<.cc><.f>	b,b,c	00101bbb11110101FBBBCCCCC0QQQQQ
QMACHU<.cc><.f>	b,b,u6	00101bbb11110101FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

QMPYH

Function

Quad 16x16 multiplication. For the non-DSP version, see [QMPYH](#).

Extension Group

(MPY_OPTION == 9) or (dsp_wide==true)

Operation

```
accwide = b.h3*c.h3+b.h2*c.h2 + b.h1*c.h1+b.h0*c.h0
a = accwide;
```



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”](#).

Instruction Format

op A,B,C

Syntax Example

QMPYH <.f> A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☒	= Set to the resulting sign of the accumulator
C	☐	= Unchanged
V	☒	= Cleared

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 then summed and assigned to the destination register pair and the 64-bit accumulator (`ACCH`, `ACCL`).

Flags are updated only if the set flags suffix (`.F`) is used. The sign (`N`) flag is set to the resulting sign of the accumulator. This instruction always clears the `V` flag.

Pseudo Code

```
accwide = b.h0 * c.h0 + b.h1 * c.h1 + b.h2 * c.h2 + b.h3 * c.h3;      /* QMPYH*/
if (DSP_CTRL.GE)              // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
a = accwide & 0xffff_ffff_ffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
if F {
    V_flag = 0;
    N_flag = rem & 1;
}
```

Assembly Code Example

```
QMPYH r0,r2,r4                ; quad 16x16 multiplication of
                              ; (r3,r2) and (r5,r4). The result
                              ; is stored in (r1,r0)
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.
For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
QMPYH<.f>	a,b,c	00101bbb00110000FBBBCCCCCAAAAAA
QMPYH<.f>	a,b,u6	00101bbb01110000FBBBuuuuuuAAAAAA
QMPYH<.f>	b,b,s12	00101bbb10110000FBBBssssssSSSSSS
QMPYH<.cc><.f>	b,b,c	00101bbb11110000FBBBCCCCC0QQQQQ
QMPYH<.cc><.f>	b,b,u6	00101bbb11110000FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.
A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

QMPYHF

Function

Quad 16x16 fractional multiplication; return a 16-bit saturated fractional result.

Extension Group

dsp_wide==true

Operation

```
if (cc) {
    acchi = b.h3*c.h3+b.h2*c.h2+b.h1*c.h1+b.h0*c.h0
    a.h1 = SAT16(acchi <<1);
}
```



Note

$h0, h1, h2$, and $h3$ represent the 16-bit elements of a 64-bit operand. 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”](#).

Instruction Format

op a,B,C

Syntax Example

QMPYH <.f> a,B,C

STATUS32 Flags Affected

Z	•	= Set if the result is zero
N	•	= Set to the resulting sign of the accumulator
C		= Unchanged
V	•	= Cleared

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 then summed and assigned to the 64-bit accumulator (ACCH, ACCL). A saturated 16-bit fractional result of the accumulator is returned in the destination register.

Flags are updated only if the set flags suffix (.F) is used.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* QMPYHF*/
s = 0;
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        add0 = b.h0 * c.h0 << 1;
        add1 = b.h1 * c.h1 << 1;
        add2 = b.h2 * c.h2 << 1;
        add3 = b.h3 * c.h3 << 1;
        acchi      = acc_add(0, add0+add1+add2+add3, 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, &add0);
        s |= sat((b.h1 * c.h1 << 1), 31, &add1);
        s |= sat((b.h2 * c.h2 << 1), 31, &add2);
        s |= sat((b.h3 * c.h3 << 1), 31, &add3);
        acchi      = acc_add(0, add0+add1+add2+add3, 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
    add0 = b.h0 * c.h0;
    add1 = b.h1 * c.h1;
    add2 = b.h2 * c.h2;
    add3 = b.h3 * c.h3;
    if (DSP_CTRL.GE)
        // guard bits are enabled
        acchi      = acc_add(0, add0+add1+add2+add3, 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, add0+add1+add2+add3, 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

```
QMPYHF r0,r2,r4    ; Signed fractional multiply quad vectors {r2,r3}
                   ;and {r4,r5} and return result in r0
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
QMPYHF<.f>	a,b,c	00101bbb00111100FBBBCCCCCAAAAA
QMPYHF<.f>	a,b,u6	00101bbb01111100FBBBuuuuuuAAAAA
QMPYHF<.f>	b,b,s12	00101bbb10111100FBBBssssssSSSSSS
QMPYHF<.cc><.f>	b,b,c	00101bbb11111100FBBBCCCCC0QQQQQ
QMPYHF<.cc><.f>	b,b,u6	00101bbb11111100FBBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

QMPYHU

Function

Quad unsigned 16x16 multiply. For the non-DSP version, see [QMPYHU](#).

Extension Group

(MPY_OPTION == 9) or (dsp_wide==true)

Operation

```
accwide = b.h3*c.h3+b.h2*c.h2 + b.h1*c.h1+b.h0*c.h0
a = accwide;
```



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"](#).

Instruction Format

op A,B,C

Syntax Example

QMPYHU <.f> A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☒	= Set if an overflow occurs during accumulation.

Description

The `B` and `C` operands specify 64-bit values, each containing four unsigned 16-bit elements. Each pair of four elements in `B` and `C` is multiplied to form four 32-bit unsigned products. The four products are then summed and assigned to the destination register pair and the 64-bit accumulator (`ACCH`, `ACCL`).

Flags are updated only if the set flags suffix (`.F`) is used. This instruction always clears the overflow (`V`) flag.

Pseudo Code

```
accwide = b.h0 * c.h0 + b.h1 * c.h1 + b.h2 * c.h2 + b.h3 * c.h3;      /* QMPYHU*/
if (DSP_CTRL.GE)              // only if guard bits are enabled
    ACC0_GHI = accwide >> 64;
a = accwide & 0xffff_ffff_ffff_ffff;
ACC0_LO = accwide & 0xffff_ffff;
ACC0_HI = (accwide >> 32) & 0xffff_ffff;
if F {
    V_flag = 0;
}
```

Assembly Code Example

```
QMPYHU r0,r2,r4      ; quad 16x16 unsigned multiplication of (r3,r2) and
                    ; (r5,r4). The results is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Long Immediate and Null Destination Operands

Instruction Code		
QMPYHU<.f>	a,b,c	00101bbb00110001FBBBCCCCCAAAAAA
QMPYHU<.f>	a,b,u6	00101bbb01110001FBBBuuuuuuAAAAAA
QMPYHU<.f>	b,b,s12	00101bbb10110001FBBBsssssssSSSSSS
QMPYHU<.cc><.f>	b,b,c	00101bbb11110001FBBBCCCCC0QQQQQ
QMPYHU<.cc><.f>	b,b,u6	00101bbb11110001FBBBuuuuuu1QQQQQ

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

REM

Function

2's complement integer Remainder. For the non-DSP version, see [REM](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

if (cc) then $a = b \% c$;

Instruction Format

op a, b, c

Syntax Example

REM <.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V	•	= Set if divisor is zero or if an arithmetic overflow occurs

Description

If the divisor (c) is non-zero, the destination register is assigned the remainder obtained by signed integer division of source operand (b) by source operand (c).

If the divisor (c) is zero, the destination register is never updated.

If the STATUS32.DZ bit is set to 1 when the divisor is zero, an EV_DivZero exception is raised. In this case, the arithmetic flags are not modified, regardless of the flag enable (F) bit for this instruction.

If the STATUS32.DZ bit is set to 0 when the divisor is zero, and if the F bit is set to 1, the Overflow flag (V) is set to 1 to indicate an attempted division by zero, but no exception is raised.

An arithmetic overflow is signaled if the quotient of (b / c) cannot be represented in 32 bits. This overflow only occurs in one specific case, which is the division of the largest representable negative value (0x80000000) by -1. If an arithmetic overflow occurs, the overflow flag (V) is set to 1.

If a division-by-zero is attempted, or if the result cannot be represented in 32 bits, when the EV_DivZero exception is disabled, the overflow flag is set to 1 and the result value is implementation-dependent. For further information on the result value returned in this case, see [Appendix Table B-1, "Implementation Behavior on Division Overflow"](#).

Pseudo Code

```

if (cc == true) {
    if ((src2 != 0) && ((src1 != 0x80000000) || (src2 != 0xffffffff)))
    {
        dest = src1 % src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = dest[31];
            V = 0;
        }
    } else {
        if ((src2 == 0) && (STATUS32.DZ == 1))
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1) {
                V = 1;
            }
        }
    }
}
}

```

Assembly Code Example

```
REM r1,r2,r3 ; r1 is assigned r2 % r3
```

Syntax and Encoding

Instruction Code

REM<.f>	a,b,c	00101bbb00001000FBBBCCCCCAAAAAA
REM<.cc><.f>	b,b,c	00101bbb11001000FBBBCCCCC0QQQQQ
REM<.cc><.f>	b,b,limm	00101bbb11001000FBBB1111100QQQQQ
REM<.cc><.f>	b,b,u6	00101bbb11001000FBBBuuuuuu1QQQQQ
REM<.f>	b,b,s12	00101bbb10001000FBBBssssssSSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

REMU

Function

Unsigned integer Remainder. For the non-DSP version, see [REMU](#).

Extension Group

DIV_REM_OPTION==radix4_enhanced or DSP_DIVSQRT==true

Operation

if (cc) 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, see [Appendix Table B-1, “Implementation Behavior on Division Overflow”](#).

If the flag-update bit is set, an unsigned REMU operation always clears the N-flag.

Pseudo Code

```

if (cc == true) {
    if (src2 != 0) {
        dest = src1 % src2;
        if (F == 1) {
            Z = (dest == 0) ? 1 : 0;
            N = 0;
            V = 0;
        }
    } else {
        if (STATUS32.DZ == 1)
            RaiseException (EV_DivZero);
        else {
            /* optional implementation-dependent assignment to dest */
            if (F == 1) {
                V = 1;
            }
        }
    }
}
}
/* REMU */

```

Assembly Code Example

```

REMU r1,r2,r3                ; r1 is assigned r2 % r3

```

Syntax and Encoding

Instruction Code

REMU<.f>	a,b,c	00101 bbb 00 001001 F BBB CCCCC AAAAAA
REMU<.f>	a,b,u6	00101 bbb 01 001001 F BBB uuuuuu AAAAAA
REMU<.cc><.f>	b,b,c	00101 bbb 11 001001 F BBB CCCCC 0QQQQQ
REMU<.cc><.f>	b,b,u6	00101 bbb 11 001001 F BBB uuuuuu 1QQQQQ
REMU<.f>	b,b,s12	00101 bbb 10 001001 F BBB ssssss SSSSSS

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

RNDH

Function

Round and saturate a 32-bit value to a 16-bit value

Extension Group

DSP basic arithmetic

Operation

$$b = \text{SAT16}(\text{RND16}(c));$$

Instruction Format

op b,c

Syntax Example

RNDH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if the result is negative
C		= Set if the result saturates
V	•	= Saturation occurs when 0x7F_FFFF is rounded.

Description

Round a 32-bit input fractional value to a 16-bit fractional value. Saturate if rounding causes an overflow of the result. The saturation flag DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. Flag updates occur only if the set flags suffix (.F) is used.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
s = sat(round(c,16),31,&t); /* RNDH */
DSP_CTRL.SAT |= s;
b = (t>>16);
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = (b & 0x8000) == 0x8000;
    STATUS32.V = s;
}
```

Assembly Code Example

```
RNDH r1,r2    ; Round 32-bits in r2 and return saturated value into
               ;r1.
```

Syntax and Encoding

Instruction Code

RNDH<.f>	b,c	00101 bbb 00 101111 F BBB CCCCC 000011
RNDH<.f>	b,limm	00101 bbb 00 101111 F BBB 111110 000011
RNDH<.f>	0,c	00101 110 00 101111 F 111 CCCCC 000011
RNDH<.f>	0,limm	00101 110 00 101111 F 111 111110 000011
RNDH<.f>	b,u6	00101 bbb 01 101111 F BBB uuuuuu 000011
RNDH<.f>	0,u6	00101 110 01 101111 F 111 uuuuuu 000011

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SATH

Function

Saturate a 32-bit value to a 16-bit value

Extension Group

DSP basic arithmetic

Operation

$b = \text{SAT16}(c)$;

Instruction Format

op b,c

Syntax Example

SATH<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if input is zero
N	•	= Set if most-significant bit of input is set
C		= Unchanged
V	•	= Set if input is smaller than 0xffff_8000 or bigger than 0x0000_7fff; otherwise cleared

Description

Saturate a 32-bit input fractional value to a 16-bit fractional value. Saturate if bit 15 up to 31 are not equal to the sign bit. The saturation flag DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix. Flag updates occur only if the set flags suffix (.F) is used.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
s = sat(c, 15, &b); /* SATH
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = b == 0x0000_0000;
    STATUS32.N = b < 0;
    STATUS32.V = s;
}
```

Assembly Code Example

```
SATH r1,r2    ; Saturate 32-bits in r2 and return saturated value
               ;into r1.
```

Syntax and Encoding

Instruction Code		
SATH<.f>	b,c	00101 bbb 00101111 FBBB CCCCC000010
SATH<.f>	b,limm	00101 bbb 00101111 FBBB 111110000010
SATH<.f>	0,c	00101 110 00101111 F111 CCCCC000010
SATH<.f>	0,limm	00101 110 00101111 F111 111110000010
SATH<.f>	b,u6	00101 bbb 01101111 FBBB uuuuuu000010
SATH<.f>	0,u6	00101 110 01101111 F111 uuuuuu000010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SATF

Function

Saturate a register based on the STATUS32.N and STATUS32.V flags.

Extension Group

DSP basic arithmetic

Operation

$$b = V ? (N ? 0x00000000 : 0xffffffff) : c;$$

Instruction Format

op b,c

Syntax Example

SATF<.f> b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

If the STATUS32.V flag is set, saturate the input operand based on the STATUS32.N flag value. If the STATUS32.N flag is set, saturate the input operand to a positive value, that is saturate the register with zeros (0s). If the STATUS32.N flag is not set, saturate the input operand to the negative value, that is saturate the register with ones (1s).

Pseudo Code

```

b = c                                     /* SATF
if (STATUS.V) {
    if (STATUS.N) {
        // saturate to positive
        b = 0x00000000;
    } else {
        // saturate to negative
        b = 0xffffffff;
    }
}

```

Assembly Code Example

```
SATF r0,r1    ; Saturate register r1 based on the N and V flags and
               ; return r0
```

Syntax and Encoding

Instruction Code

SATF<.f>	b, c	00101 bbb 00 101111 FBBB CCCCC 011010
SATF<.f>	b, u6	00101 bbb 01 101111 FBBB uuuuuu 011010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SBCS

Function

Signed subtraction with saturation and carry in

Extension Group

DSP basic arithmetic

Operation

$$a = \text{SAT32}((b - c) - \text{STATUS32.C});$$

Instruction Format

op a, b, c

Syntax Example

SBCS<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Cleared
V	•	= Set if the result is saturated

Description

Subtract source operand 2 (c) from source operand 1 (b), and also subtract the state of the carry flag STATUS32.C. Store the saturated result in the destination register (a). If the carry flag is set after the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used. If the subtraction overflows, the result is saturated to the maximum positive value 0x7FFF_FFFF or the minimum negative value 0x8000_0000. The saturation flag DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix.

Pseudo Code

```
s = sat((sint128) (b - c) - STATUS32.C, 31, &a);      /* SBCS */
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
    STATUS32.C = 0;
}
```

Assembly Code Example

```
SBCS r0,r1,r2      ; Subtract with carry-in r1 and r2 and write the
                   ; saturated result into r0
```

Syntax and Encoding

Instruction Code

SBCS<.f>	a,b,c	00101bbb00100111FBBBCCCCC	AAAAAA
SBCS<.f>	a,b,u6	00101bbb01100111FBBBuuuuuu	AAAAAA
SBCS<.f>	b,b,s12	00101bbb10100111FBBBssssss	SSSSSS
SBCS<.cc><.f>	b,b,c	00101bbb11100111FBBBCCCCC	0QQQQQ
SBCS<.cc><.f>	b,b,u6	00101bbb11100111FBBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SETACC

Function

Set the accumulator value.

Extension Group

DSP Accumulator

Operation

```
if (c & 0x1) {
  switch (c.b1) {
    0: accwide = (int) b;
    1: acclo = (int) b;
    2: acchi = (int) b;
    3: acchi = (int) b; acclo = (int) b;
    4: accwide = (fract) b;
    5: acclo = (fract) b;
    6: acchi = (fract) b;
    7: acclo = (fract) b; acchi = (fract) b;
    8: accwide = (unsigned) b;
    9: acclo = (unsigned) b;
    10: acchi = (unsigned) b;
    11: acclo = (unsigned) b; acchi = (unsigned) b;
  }
} else {
  switch (c.b1) {
    0: ACC0_LO = b;
    1: ACC0_HI = b;
    2: ACC0_GLO.GUARD = b;
    3: ACC0_GHI.GUARD = b;
  }
}
```

Instruction Format

op a,b,c

Syntax Example

SETACC a,b, c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Set a value in the accumulator auxiliary registers with optional sign extension. The return register is ignored.

Although the return value is ignored, using the loop-counter (LP_COUNT/r60) or the program counter (PCL/r63) as a destination operand to the SETACC instruction results in an illegal instruction exception.

The SETACC instruction ignores the destination operand, similar to an SR instruction.

Instruction Encoding

Operand c	Description
Bits [0]	0: One of the auxiliary registers is initialized 1: SETACC sign extends the value and the accumulator including the guard bits are initialized
Bits[9:8]	Denotes the accumulator or the accumulator register to be used in the operation 0: initialize wide accumulator with integer 1: low accumulator 2: high accumulator 3: initialize wide accumulator with a fractional value

Pseudo Code

```

if (c & 0x1) { /* 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	00101 bbb 00 0011011 BBB CCCCC AAAAAA
SETACC	a, b, u6	00101 bbb 01 0011011 BBB uuuuuu AAAAAA
SETACC	b, b, s12	00101 bbb 10 0011011 BBB ssssss SSSSSS
SETACC<.cc>	b, b, c	00101 bbb 11 0011011 BBB CCCCC 0QQQQQ
SETACC<.cc>	b, b, u6	00101 bbb 11 0011011 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SQRT

Function

Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result

Extension Group

DSP_DIVSQRT!= none

Operation

```
if (cc) {
    acchi = sqrt(c);
    acclo = c - sqrt(c)2;
    b = acchi;
}
```

Instruction Format

op b, c

Syntax Example

SQRT<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Compute the unsigned square-root of a 32-bit integer, yielding a 16-bit result. The flags are updated only if the .F suffix is specified.

Pseudo Code

```
if (cc == true) {
    ACC0_HI = sqrt(c);
    ACC0_LO = c - ACC0_HI*ACC0_HI;
    b = ACC0_HI;
    if (F == 1) {
        Z_flag = (b == 0) ? 1 : 0;
    }
}
```

/* SQRT */

Assembly Code Example

```
SQRT r1,r2      ; Unsigned integer square-root of r2
                ; return result in r1
```

Syntax and Encoding

Instruction Code		
SQRT<.f>	b,c	00101bbb001011110BBBCCCCC110000
SQRT<.f>	b,u6	00101bbb011011110BBBuuuuuu110000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SQRTF

Function

Compute the square-root of an unsigned Q31 fraction

Extension Group

DSP_DIVSQRT!= none

Operation

```
if (cc) {
    acchi = sqrt(c<<31);
    acclo = (c<<31) - sqrt(c)2;
    b = acchi;
}
```

Instruction Formats

op b, c

Syntax Example

SQRTF<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Compute the square-root of an unsigned Q31 fraction, yielding a Q31 fractional result. The flags are updated only if the .F suffix is specified.

Pseudo Code

```

if (cc == true) {                                     /* SQRTF */
    i = c<<31; // 64b unsigned
    ACC0_HI = sqrt(i);
    ACC0_LO = (i) - ACC0_HI*ACC0_HI;
    b = ACC0_HI;
    if (F == 1) {
        Z_flag = (b == 0) ? 1 : 0;
    }
}

```

Assembly Code Example

```

SQRTF r1,r2      ; Fractional square-root of r2
                  ; return result in r1

```

Syntax and Encoding

Instruction Code

SQRTF<.f>	b, c	00101 bbb 00 101111 0 BBB CCCCC 110001
SQRTF<.f>	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 110001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

SUBS

Function

Signed 32-bit subtraction; the result is saturated

Extension Group

DSP basic arithmetic

Operation

$$a = \text{SAT32}(b - c)$$

Instruction Format

op a, b, c

Syntax Example

SUBS<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Cleared
V	•	= Set if result is saturated

Description

Obtain the saturating difference of long word operands b and c, and place the result in the destination register a. If the subtraction overflows, the result is saturated to the maximum positive value 0x7fff_ffff or the minimum negative value 0x8000_0000. The saturation flag DSP_CTRL.SAT is set if the result of the instruction saturates, regardless of the .F suffix.

If the set flags suffix (.F) is used, the other flags are updated. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
s = sat((sint128) b - c, 31, &a);          /* SUBS */
DSP_CTRL.SAT |= s;
if (F) {
    STATUS32.Z = a == 0x0000_0000;
    STATUS32.N = (a & 0x8000_0000) != 0;
    STATUS32.V = s;
    STATUS32.C = 0;
}
```

Assembly Code Example

```
SUBS r0,r1,r2    ; Subtract r2 from r1 and write saturated result into r1
```

Syntax and Encoding

Instruction Code

SUBS<.f>	a,b,c	00101 bbb 00 000111 FBBBCCCCC AAAAAA
SUBS<.f>	a,b,u6	00101 bbb 01 000111 FBBBuuuuuu AAAAAA
SUBS<.f>	b,b,s12	00101 bbb 10 000111 FBBBssssss SSSSSS
SUBS<.cc><.f>	b,b,c	00101 bbb 11 000111 FBBBCCCCC 0QQQQQ
SUBS<.cc><.f>	b,b,u6	00101 bbb 11 000111 FBBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VABS2H

Function

Absolute value of a two-way 16-bit vector.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = ABS(c.h1);
b.h0 = ABS(c.b0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VABS2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Obtain the pair-wise absolute value of a two-way vector operand, *c*, and place the result in the destination register, *b*. This instruction does not modify any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
l = (c & 0x0000_ffff); /* VABS2H*/
h = ((c>>16) & 0x0000_ffff);
l = ((l & 0x8000) != 0) ? -l : l;
h = ((h & 0x8000) != 0) ? -h : h;
b = (h << 16) | (l & 0x0000_ffff);
```

Assembly Code Example

```
VABS2H r1,r2      ; Absolute value of a two-way 16-bit vector (r2) is
                  ; stored in r1.
```

Syntax and Encoding

Instruction Code		
VABS2H	b, c	00101 bbb 00 101111 0 BBB CCCCC 101000
VABS2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101000
VABS2H	0, c	00101 110 00 101111 0 111 CCCCC 101000
VABS2H	0, limm	00101 110 00 101111 0 111 111110 101000
VABS2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101000
VABS2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VABSS2H

Function

Saturated absolute value of a two-way 16-bit vector.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = SAT16 (ABS (c.h1)) ;
b.h0 = SAT16 (ABS (c.b0)) ;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VABSS2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Obtain the pair-wise absolute value of a two-way vector operand, *c*, and place the saturated result in the destination register *b*. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not modify any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
l = (c & 0x0000_ffff); /* VABSS2H*/
h = ((c>>16) & 0x0000_ffff);
s = (l == 0x8000) || (h == 0x8000);
l = l == 0x8000 ? 0x7fff : (((l & 0x8000) != 0) ? -1 : 1);
h = h == 0x8000 ? 0x7fff : (((h & 0x8000) != 0) ? -h : h);
b = (h << 16) | (l & 0x0000_ffff);
DSP_CTRL.SAT |= s
```

Assembly Code Example

```
VABSS2H r1,r2      ; Saturated absolute value of a two-way 16-bit
                   vector (r2) is ;stored in r1.
```

Syntax and Encoding

Instruction Code

VABSS2H	b, c	00101 bbb 00 101111 0 BBB CCCCC 101001
VABSS2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101001
VABSS2H	0, c	00101 110 00 101111 0 111 CCCCC 101001
VABSS2H	0, limm	00101 110 00 101111 0 111 111110 101001
VABSS2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101001
VABSS2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADD2H

Function

Dual 16-bit SIMD addition

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 + c.h1;
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about the vector operands, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADD2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit `b` and `c` operands specify vectors containing two 16-bit elements. Each pair of elements from `b` and `c` is added to form two 16-bit sums. These are assigned to the destination register, if defined.

This instruction does not modify any flags.

An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b1 = b & 0x0000_ffff; /* VADD2H*/
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
a1 = b1 + cl;
ah = bh + ch;
a = (a1 & 0x0000_ffff) | (ah << 16);
```

Assembly Code Example

```
VADD2H r1,r2,r3      ; dual SIMD 16-bit addition of r2 and r3
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
VADD2H	a,b,c	00101bbb000101000BBBCCCCC0AAAAAA
VADD2H	a,b,u6	00101bbb010101000BBBuuuuuu0AAAAAA
VADD2H	b,b,s12	00101bbb100101000BBBsssssssSSSSSS
VADD2H<.cc>	b,b,c	00101bbb110101000BBBCCCCC0QQQQQ
VADD2H<.cc>	b,b,u6	00101bbb110101000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADD4B

Function

Four-way 8-bit vector addition.

Extension Group

DSP vector arithmetic

Operation

$$\begin{aligned} a.b3 &= b.b3 + c.b3; \\ a.b2 &= b.b2 + c.b2; \\ a.b1 &= b.b1 + c.b1; \\ a.b0 &= b.b0 + c.b0; \end{aligned}$$


Note

$h0$ and $h1$ represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VADD4B a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Pair-wise byte addition of the b and c operands. The results are stored in the a operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

b1l = b & 0x0000_00ff;
b1h = b & 0x0000_ff00;
bhl = b & 0x00ff_0000;
bhh = b & 0xff00_0000;
c1l = c & 0x0000_00ff;
c1h = c & 0x0000_ff00;
chl = c & 0x00ff_0000;
chh = c & 0xff00_0000;
all = b1l + c1l;
alh = b1h + c1h;
ahl = bhl + chl;
ahh = bhh + chh;
a = (all & 0x0000_00ff) | (alh & 0x0000_ff00) | (ahl & 0x00ff_0000) |
    ahh;

```

/* VADD4B/*

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	00101bbb001001000BBBCCCCCAAAAAA
VADD4B	a, b, u6	00101bbb011001000BBBuuuuuuAAAAAA
VADD4B	b, b, s12	00101bbb101001000BBBssssssSSSSSS
VADD4B<.cc>	b, b, c	00101bbb111001000BBBCCCCC0QQQQQ
VADD4B<.cc>	b, b, u6	00101bbb111001000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDS2

Function

Two-way 32-bit vector addition.

Extension Group

dsp_wide==true

Operation

```
A.w1 = SAT32 (B.w1+C.w1) ;
A.w0 = SAT32 (B.w0+C.w0) ;
```

Instruction Format

op a,b,c

Syntax Example

VADDS2 A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Pair-wise 32-bit addition of the B and C operands. The saturated results are stored in the A operand.

Pseudo Code

```
s = sat((sint128) b.w0 + c.w0, 31, &a.w0) | /* VADDS2/*
sat((sint128) b.w1 + c.w1, 31, &a.w1);
DSP_CTRL.SAT |= s;
```

Assembly Code Example

```
VADDS2H r0,r2,r4      ; Add {r2,r3} and {r4,r5} and write result into {r0,r1}
                       ; while saturating
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

		Instruction Code
VADDS2	a, b, c	00101bbb001111001BBBCCCCCAAAAAA
VADDS2	a, b, u6	00101bbb011111001BBBuuuuuuAAAAAA
VADDS2	b, b, s12	00101bbb101111001BBBssssssSSSSSS
VADDS2<.cc>	b, b, c	00101bbb111111001BBBCCCCC0QQQQQ
VADDS2<.cc>	b, b, u6	00101bbb111111001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADD2

Function

Dual 32-bit addition. For the non-DSP version, see [VADD2](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.w0 = B.w0 + C.w0;
    A.w1 = B.w1 + C.w1;}
```



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"](#).

Instruction Format

op A,B,C

Syntax Example

VADD2 A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing two 32-bit elements. These elements are normally register pairs, although the operand formats are not restricted to specifying registers. Each pair of elements from B and C is added to form two 32-bit sums. These are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if (cc)==true then                                     /* VADD2*/
    A.w0 = B.w0 + C.w0
    A.w1 = B.w1 + C.w1
```

Assembly Code Example

```
VADD2 r0,r2,r4    ; dual SIMD 32-bit addition of (r5,r4) and (r3,r2).
                  ;the result is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
VADD2	a,b,c	00101bbb001111000BBBCCCCCAAAAAA
VADD2	a,b,u6	00101bbb011111000BBBuuuuuuAAAAAA
VADD2	b,b,s12	00101bbb101111000BBBsssssssSSSSSS
VADD2<.cc>	b,b,c	00101bbb111111000BBBCCCCC0QQQQQ
VADD2<.cc>	b,b,u6	00101bbb111111000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADD4H

Function

Quad 16-bit SIMD addition. For the non-DSP version, see [VADD4H](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.h0 = B.h0 + C.h0;
    A.h1 = B.h1 + C.h1;
    A.h2 = B.h2 + C.h2;
    A.h3 = B.h3 + C.h3;
}
```



Note

$h0, h1, h2, h3$ represent the 16-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"](#).

Instruction Format

op A,B,C

Syntax Example

VADD4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each pair of elements from B and C is added to form four 16-bit sums. These 16-bit sums are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if (cc)== true then                                     /* VADD4H*/
    A.h0 = B.h0 + C.h0
    A.h1 = B.h1 + C.h1
    A.h2 = B.h2 + C.h2
    A.h3 = B.h3 + C.h3
```

Assembly Code Example

```
VADD4H r0,r2,r4    ; quad SIMD 16-bit addition of (r3,r2) and (r5,r4). The
                   ;result is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
VADD4H	a,b,c	00101bbb001110000BBBCCCCCAAAAAA
VADD4H	a,b,u6	00101bbb011110000BBBuuuuuuAAAAAA
VADD4H	b,b,s12	00101bbb101110000BBBsssssssSSSSSS
VADD4H<.cc>	b,b,c	00101bbb111110000BBBCCCCC0QQQQQ
VADD4H<.cc>	b,b,u6	00101bbb111110000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUB

Function

Dual 32-bit SIMD add and subtract. For the non-DSP version, see [VADDSUB](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.w0 = B.w0 + C.w0;
    A.w1 = B.w1 - C.w1;
}
```



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”](#).

Instruction Format

op A,B,C

Syntax Example

VADDSUB A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing two 32-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. The first pair of elements from B and C is added to form a 32-bit sum. The second pair of elements is subtracted to form a 32-bit difference. The sum and difference are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {                                     /* VADDSUB*/
    A.w0 = B.w0 + C.w0;
    A.w1 = B.w1 - C.w1;}

```

Assembly Code Example

```
VADDSUB r0,r2,r4      ; dual SIMD 32-bit addition and subtraction of (r3,r2)
                     ;and (r5,r4). The result is stored in (r1,r0)

```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
VADDSUB	a,b,c	00101bbb001111100BBBCCCCCAAAAAA
VADDSUB	a,b,u6	00101bbb011111100BBBuuuuuuAAAAAA
VADDSUB	b,b,s12	00101bbb101111100BBBsssssssSSSSSS
VADDSUB<.cc>	b,b,c	00101bbb111111100BBBCCCCC0QQQQQ
VADDSUB<.cc>	b,b,u6	00101bbb111111100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUB2H

Function

Dual 16-bit SIMD add and subtract. For the non-DSP version, see [VADDSUB2H](#).

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit *b* and *c* operands specify vectors containing two 16-bit elements. The higher 16 bits from *b* and *c* is added to form a 16-bit sum. The lower 16 bits are subtracted to form a 16-bit difference. The sum and difference are assigned to the destination register, if defined.

This instruction does not modify any flags. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if(cc) {                                     /* VADDSUB2H*/
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```

Assembly Code Example

```
VADDSUB2H r1,r2,r3      ; dual SIMD 16-bit addition and subtraction
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
VADDSUB2H	a,b,c	00101bbb000101100BBBCCCCCAAAAAA
VADDSUB2H	a,b,u6	00101bbb010101100BBBuuuuuuAAAAAA
VADDSUB2H	b,b,s12	00101bbb100101100BBBsssssssSSSSSS
VADDSUB2H<.cc>	b,b,c	00101bbb110101100BBBCCCCC0QQQQQ
VADDSUB2H<.cc>	b,b,u6	00101bbb110101100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUB4H

Function

Quad 16-bit SIMD add and subtract. For the non-DSP version, see [VADDSUB4H](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.h0 = B.h0 + C.h0;
    A.h1 = B.h1 - C.h1;
    A.h2 = B.h2 + C.h2;
    A.h3 = B.h3 - C.h3;
}
```



Note

$h0$, $h1$, $h2$, and $h3$ represent the 16-bit elements of a 32-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"](#).

Instruction Format

op A,B,C

Syntax Example

VADDSUB4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each even-numbered pair of elements from B and C is added to form 16-bit sums. Each odd-numbered pair of elements from B and C is subtracted to form 16-bit differences. The four elemental results are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {                                     /* VADDSUB4H*/
    A.h0 = B.h0 + C.h0;
    A.h1 = B.h1 - C.h1;
    A.h2 = B.h2 + C.h2;
    A.h3 = B.h3 - C.h3;
}
```

Assembly Code Example

```
VADDSUB4H r0,r2,r4    ; quad SIMD 16-bit addition and subtraction of
                      ; (r3,r2) and (r5,r4). The result is stored in
                      ; (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
VADDSUB4H	a,b,c	00101bbb001110100BBBCCCCCAAAAAA
VADDSUB4H	a,b,u6	00101bbb011110100BBBuuuuuuAAAAAA
VADDSUB4H	b,b,s12	00101bbb101110100BBBssssssSSSSSS
VADDSUB4H<.cc>	b,b,c	00101bbb111110100BBBCCCCC0QQQQQ
VADDSUB4H<.cc>	b,b,u6	00101bbb111110100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,r0,r1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDS2H

Function

Two-way 16-bit vector addition.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = SAT16 (b.h1+c.h1) ;
a.h0 = SAT16 (b.h0+c.h0) ;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VADDS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Pair-wise 16-bit addition of the *b* and *c* operands. The saturated results are stored in the *a* operand. If any of the addition results saturate, the saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                     /* VADDS2H/*
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)b1 + cl, 15, al);
sh = sat((sint128)bh + ch, 15, ah);
a = (al & 0x0000_ffff) | (ah << 16);
DSP_CTRL.SAT |= s1 || 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	00101bbb000101001BBBCCCCCAAAAAA
VADDS2H	a,b,u6	00101bbb010101001BBBuuuuuuAAAAAA
VADDS2H	b,b,s12	00101bbb100101001BBBssssssSSSSSS
VADDS2H<.cc>	b,b,c	00101bbb110101001BBBCCCCC0QQQQQ
VADDS2H<.cc>	b,b,u6	00101bbb110101001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDS4H

Function

Four-way 16-bit vector addition.

Extension Group

dsp_wide==true

Operation

```
A.h3 = SAT16 (B.h3+C.h3) ;
A.h2 = SAT16 (B.h2+C.h2) ;
A.h1 = SAT16 (B.h1+C.h1) ;
A.h0 = SAT16 (B.h0+C.h0) ;
```



Note

h0, h1, h2, and h3 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VADDS4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Pair-wise 16-bit addition of the B and C operands. The saturated results are stored in the A operand. If any of the addition results saturate, the saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
s = sat((sint128)b.h0 + c.h0, 15, &a.h0) | /* VADDS4H/*
      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

```
VADDS4H r0,r2,r4      ; Add 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](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
VADDS4H	a,b,c	00101bbb0011110001BBBCCCCCAAAAAA
VADDS4H	a,b,u6	00101bbb0111110001BBBuuuuuuAAAAAA
VADDS4H	b,b,s12	00101bbb1011110001BBBssssssSSSSSS
VADDS4H<.cc>	b,b,c	00101bbb1111110001BBBCCCCC0QQQQQ
VADDS4H<.cc>	b,b,u6	00101bbb1111110001BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUB2H

Function

Dual 16-bit SIMD add and subtract. For the non-DSP version, see [VADDSUB2H](#).

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = b.h0 + c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit *b* and *c* operands specify vectors containing two 16-bit elements. The higher 16 bits from *b* and *c* is added to form a 16-bit sum. The lower 16 bits are subtracted to form a 16-bit difference. The sum and difference are assigned to the destination register, if defined.

This instruction does not modify any flags.

An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                /* VADDSUB2H*/
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
a1 = b1 + cl;
ah = bh - ch;
a = (a1 & 0x0000_ffff) | (ah << 16);

```

Assembly Code Example

```
VADDSUB2H r1,r2,r3      ; dual SIMD 16-bit addition and subtraction
```

Syntax and Encoding



Note

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code

VADDSUB2H	a, b, c	00101 bbb 00 010110 0BBBCCCCC AAAAAA
VADDSUB2H	a, b, u6	00101 bbb 01 010110 0BBBuuuuuu AAAAAA
VADDSUB2H	b, b, s12	00101 bbb 10 010110 0BBBssssss SSSSSS
VADDSUB2H<.cc>	b, b, c	00101 bbb 11 010110 0BBBCCCCC 0QQQQQ
VADDSUB2H<.cc>	b, b, u6	00101 bbb 11 010110 0BBBuuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUBS

Function

Two-way 32-bit vector addition and subtraction. The results are saturated and stored.

Extension Group

dsp_wide==true

Operation

```
if (cc) {
    a.w0 = SAT32 (b.w0 + c.w0);
    a.w1 = SAT32 (b.w1 - c.w1);
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUBS a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The `b` and `c` operands are 64-bit operands. The higher 32-bits from `b` and `c` is added to form a 32-bit sum. The lower 32 bits are subtracted to form a 32-bit difference. The saturated results are assigned to the lower and upper half-words of the destination register.

Pseudo Code

```
s = sat((sint128) b.w0 + c.w0, 31, &a.w0) | /* VADDSUBS*/
sat((sint128) b.w1 - c.w1, 31, &a.w1);
DSP_CTRL.SAT |= s;
```

Assembly Code Example

```
VADDSUBS r0,r2,r4      ; dual SIMD 32-bit addition and subtraction and
                        ;store the saturated result
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VADDSUBS	a, b, c	00101bbb001111101BBBCCCCCAAAAAA
VADDSUBS	a, b, u6	00101bbb011111101BBBuuuuuuAAAAAA
VADDSUBS	b, b, s12	00101bbb101111101BBBssssssSSSSSS
VADDSUBS<.cc>	b, b, c	00101bbb111111101BBBCCCCC0QQQQQ
VADDSUBS<.cc>	b, b, u6	00101bbb111111101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUBS2H

Function

Two-way 16-bit vector addition and subtraction. The results are saturated and stored.

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = SAT16 (b.h0 + c.h0);
    a.h1 = SAT16 (b.h1 - c.h1);
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUBS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit `b` and `c` operands specify vectors containing two 16-bit elements. The higher 16 bits from `b` and `c` is added to form a 16-bit sum. The lower 16 bits are subtracted to form a 16-bit difference. The saturated results are assigned to the lower and upper half-words of the destination register.

The saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix if the results of this instruction saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b1 = b & 0x0000_ffff; /* VADDSUBS2H*/
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)b1 + cl, 15, a1);
sh = sat((sint128)bh - ch, 15, ah);
a = (a1 & 0x0000_ffff) | (ah << 16);
DSP_CTRL.SAT |= s
```

Assembly Code Example

```
VADDSUBS2H r1,r2,r3 ; dual SIMD 16-bit addition and subtraction and
                   ;store the saturated result
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
VADDSUBS2H	a, b, c	00101bbb000101101BBBCCCCCAAAAAA
VADDSUBS2H	a, b, u6	00101bbb010101101BBBuuuuuuAAAAAA
VADDSUBS2H	b, b, s12	00101bbb100101101BBBssssssSSSSSS
VADDSUBS2H<.cc>	b, b, c	00101bbb110101101BBBCCCCC0QQQQQ
VADDSUBS2H<.cc>	b, b, u6	00101bbb110101101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands. Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VADDSUBS4H

Function

Four-way 16-bit vector addition and subtraction. The results are saturated and stored.

Extension Group

dsp_wide==true

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 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VADDSUBS4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. Each operand contains 16-bit elements (h0 through h3). The h0 and h2 elements of the C operand are added to the respective elements of the B operand, and the saturated result is stored in the respective elements of the destination operand. The h1 and h3 elements of the C operand are subtracted from the respective elements of the B operand and the saturated result is stored in the respective elements of the destination operand.

The saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix if any of the results of this instruction saturate. 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) | /* VADDSUBS4H*/
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

```
VADDSUBS4H r0,r2,r4    ; Add/subtract four vectors {r2,r3} and {r4,r5} and
                        ; return saturated result in {r0,r1}
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see [Vector Operands](#). For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
VADDSUBS4H	a,b,c	00101bbb001110101BBBCCCCCAAAAAA
VADDSUBS4H	a,b,u6	00101bbb011110101BBBuuuuuuAAAAAA
VADDSUBS4H	b,b,s12	00101bbb101110101BBBsssssssSSSSSS
VADDSUBS4H<.cc>	b,b,c	00101bbb111110101BBBCCCCC0QQQQQ
VADDSUBS4H<.cc>	b,b,u6	00101bbb111110101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts `r1` from `r0`, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

VALGN2H

Function

Compose a dual 16-bit vector from the least-significant 16 bits of the *b* operand and the most-significant 16 bits of the *c* operand.

Extension Group

DSP vector unpacking

Operation

```
{
  a.h1 = b.h0;
  a.h0 = c.h1;
}
```



Note

h0 and *h1* represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VALGN2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector from the least-significant 16 bits of the *b* operand and the most-significant 16 bits of the *c* operand. This instruction is used for realigning vector data before load and store operations. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
a.h1 = b.h0
a.h0 = c.h1;                                     /* VALGN2H*/
```

Assembly Code Example

```
VALGN2H r0,r1,r2      ; compose a vector from lower 16-bits of r1 and
                      ;higher 16-bits of r2
```

Syntax and Encoding

		Instruction Code
VALGN2H	a,b,c	00101bbb000011010BBBCCCCCAAAAAA
VALGN2H	a,b,u6	00101bbb010011010BBBuuuuuuAAAAAA
VALGN2H	b,b,s12	00101bbb100011010BBBssssssSSSSSS
VALGN2H<.cc>	b,b,c	00101bbb110011010BBBCCCCC0QQQQQ
VALGN2H<.cc>	b,b,u6	00101bbb110011010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VASL2H

Function

Perform a two-way arithmetic shift left operation 16-bit vectors. The shift amount is specified in the second operand.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = b.h1 << c.h1;
a.h0 = b.h0 << c.h0;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASL2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Left shift the 16-bit vector elements of the operand b by the two-way vector value in operand c. The shift amount is unsigned and only the bottom four bits of the vector shift amount are used. This instruction never set flags. The resulting vector is stored in the destination operand. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

bl = b & 0x0000_ffff;
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_000f;
ch = (c >> 16) & 0x0000_000f;
al = bl << cl;
ah = bh << ch;
a = (al & 0x0000_ffff) | (ah << 16);
/* VASL2H/ */

```

Assembly Code Example

```

VASL2H r0,r1,r2      ; Shift the operand r1 left by scalar r2, and return
                     ; the result to r0.

```

Syntax and Encoding

Instruction Code

VASL2H	a,b,c	00101bbb001000010BBBCCCCCAAAAAA
VASL2H	a,b,u6	00101bbb011000010BBBuuuuuuAAAAAA
VASL2H	b,b,s12	00101bbb101000010BBBssssssSSSSSS
VASL2H<.cc>	b,b,c	00101bbb111000010BBBCCCCC0QQQQQ
VASL2H<.cc>	b,b,u6	00101bbb111000010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VASLS2H

Function

Two-way 16-bit vector arithmetic shift left and store the saturated result.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = SAT16 (b.h1 << c.h1);
a.h0 = SAT16 (b.h0 << c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASLS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Left shift the 16-bit vector elements of the operand *b* by the two-way vectors in the *c* operand. The shift amount can be negative, effectively shifting right. The resulting vector is saturated and stored in the destination operand. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

If the shift amount is smaller than -15, the shift amount saturates to -15.

Pseudo Code

```

bl = b & 0x0000_ffff;                                     /* VASLS2H/*
bh = (b >> 16) & 0x0000_ffff;
cl = sext16(c & 0x0000_ffff);
ch = c >> 16;
cl = cl > -15 ? cl : -15;
ch = ch > -15 ? ch : -15;
if (cl >= 0)
    sl = sat((sint128)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	00101bbb001000011BBBCCCCCAAAAAA
VASLS2H	a,b,u6	00101bbb011000011BBBuuuuuuAAAAAA
VASLS2H	b,b,s12	00101bbb101000011BBBssssssSSSSSS
VASLS2H<.cc>	b,b,c	00101bbb111000011BBBCCCCC0QQQQQ
VASLS2H<.cc>	b,b,u6	00101bbb111000011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VASR2H

Function

Perform a two-way arithmetic shift right operation on 16-bit vectors. The shift amount is specified in the second operand.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = b.h1 >> c.h1;
a.h0 = b.h0 >> c.h0;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASR2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Right shift the 16-bit vector elements of the operand *b* by the scalar value in the *c* operand. The shift amount is unsigned and only the lower 4 bits of the vector shift amount is used. The resulting vector is stored in the destination operand. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

bl = b & 0x0000_ffff;
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_000f;
ch = (c >> 16) & 0x0000_000f;
al = bl >> cl;
ah = bh >> ch;
a = (al & 0x0000_ffff) | (ah << 16);
/* VASR2H/ */

```

Assembly Code Example

```

VASR2H r0,r1,r2      ; Shift the operand r1 right by scalar r2, and
                     ;return the result to r0.

```

Syntax and Encoding

Instruction Code

VASR2H	a,b,c	00101bbb001000100BBBCCCCC	AAAAAA
VASR2H	a,b,u6	00101bbb011000100BBBuuuuuu	AAAAAA
VASR2H	b,b,s12	00101bbb101000100BBBssssss	SSSSSS
VASR2H<.cc>	b,b,c	00101bbb111000100BBBCCCCC	0QQQQQ
VASR2H<.cc>	b,b,u6	00101bbb111000100BBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VASRS2H

Function

Perform a two-way arithmetic shift right operation on 16-bit vectors. The result is saturated.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = SAT16 (b.h1 >> c.h1);
a.h0 = SAT16 (b.h0 >> c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASRS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Right shift the 16-bit vector elements of the operand *b* by the two-way vectors in the *c* operand. The shift amount can be negative, effectively shifting left. The resulting vector is saturated and stored in the destination operand. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not update any STATUS32 flags.

If the shift amount is greater than 15, the shift amount saturates to 15.

Pseudo Code

```

bl = b & 0x0000_ffff;                                     /* VASRS2H/*
bh = (b >> 16) & 0x0000_ffff;
cl = sext16(c & 0x0000_ffff);
ch = c >> 16;
cl = cl < 15 ? cl : 15;
ch = ch < 15 ? ch : 15;
if (cl < 0)
    sl = sat((sint128)bl << -cl, 15, &al);
else
    al = bl >> cl;
if (ch < 0)
    sh = sat((sint128)bh << -ch, 15, &ah);
else
    ah = bh >> ch;
DSP_CTRL.SAT |= sl || sh
a = (al & 0x0000_ffff) | (ah << 16);

```

Assembly Code Example

```

VASRS2H r0,r1,r2      ; Shift the operand r1 right by scalar r2, and
                      ;return the saturated result to r0.

```

Syntax and Encoding

Instruction Code

VASRS2H	a,b,c	00101 bbb 00 1000101 BBBCCCCC AAAAAA
VASRS2H	a,b,u6	00101 bbb 01 1000101 BBBuuuuuu AAAAAA
VASRS2H	b,b,s12	00101 bbb 10 1000101 BBBssssss SSSSSS
VASRS2H<.cc>	b,b,c	00101 bbb 11 1000101 BBBCCCCC 0QQQQQ
VASRS2H<.cc>	b,b,u6	00101 bbb 11 1000101 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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30.

However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple *limm* values, see section Dual-operand Instructions, F32_GEN4.

VASRSR2H

Function

Perform a two-way arithmetic shift right operation on 16-bit vectors. The result is rounded and saturated.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = SAT16 (RND16 (b.h1 >> c.h1)) ;
a.h0 = SAT16 (RND16 (b.h0 >> c.h0)) ;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VASRSR2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Right shift the 16-bit vector elements of the operand *b* by the two-way vector in the *c* operand. Bits shifted out at the LSB side contributes to rounding. Rounding mode is defined in `DSP_CTRL.RM`. If the shift amount is negative, the vectors are shifted left. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the *.F* suffix.

If the shift amount is greater than 15, the shift amount saturates to 15.

This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

bl = b & 0x0000_ffff;                                     /* VASRSR2H/*
bh = (b >> 16) & 0x0000_ffff;
cl = sext16(c & 0x0000_ffff);
ch = c >> 16;
cl = cl < 16 ? cl : 16;
ch = ch < 16 ? ch : 16;
cl = cl < -16 ? -16 : cl;
ch = ch < -16 ? -16 : ch;
if (cl < 0)
    sl = sat((sint128)bl << -cl, 15, &a1);
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	00101bbb001000111BBBCCCCC	AAAAAA
VASRSR2H	a,b,u6	00101bbb011000111BBBuuuuuu	AAAAAA
VASRSR2H	b,b,s12	00101bbb101000111BBBssssss	SSSSSS
VASRSR2H<.cc>	b,b,c	00101bbb111000111BBBCCCCC	0QQQQQ
VASRSR2H<.cc>	b,b,u6	00101bbb111000111BBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VEXT2BHL

Function

Compose a dual 16-bit vector by zero-extending lower two bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = 0;
b.b2 = c.b1;
b.b1 = 0;
b.b0 = c.b0;
```



Note

b0, b1, b2, and b3 represent the 8-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VEXT2BHL b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by zero-extending the two lower bytes of the *c* operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = (c & 0x0000_00ff) | ((c & 0x0000_ff00) << 8); /* VEXT2BHL*/
```

Assembly Code Example

```
VEXT2BHL r1,r2      ; zero extend two lower bytes from r2 into 16-bit
                    ; words in r1
```

Syntax and Encoding

Instruction Code		
VEXT2BHL	b, c	00101 bbb 00 101111 0 BBB CCCCC 100100
VEXT2BHL	b, limm	00101 bbb 00 101111 0 BBB 111110 100100
VEXT2BHL	0, c	00101 110 00 101111 0 111 CCCCC 100100
VEXT2BHL	0, limm	00101 110 00 101111 0 111 111110 100100
VEXT2BHL	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100100
VEXT2BHL	0, u6	00101 110 01 101111 0 111 uuuuuu 100100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VEXT2BHLF

Function

Compose a vector with two Q15 elements by expanding the lower two Q7 bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = c.b1;
b.b2 = 0;
b.b1 = c.b0;
b.b0 = 0;
```



Note

b0, b1, b2, and b3 represent the Q7 elements of an operand.

Instruction Format

op b,c

Syntax Example

VEXT2BHLF b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

MSB-aligned expansion of the two lower bytes (Q7) of an operand into two Q15 elements. This instruction does not update any flags.

Pseudo Code

```
b = ((c & 0x0000_00ff)<<8) | ((c & 0x0000_ff00)<<16); /* VEXT2BHLF*/
```

Assembly Code Example

```
VEXT2BHLF r0,r1      ; Expand and pack 2 lower bytes from r1 into
                      ; 2 16b words in r0
```

Syntax and Encoding

Instruction Code

VEXT2BHLF	b, c	00101 bbb 00 101111 0 BBB CCCCC 100000
VEXT2BHLF	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100000

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VEXT2BHM

Function

Compose a dual 16-bit vector by zero-extending the higher two bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = 0;
b.b2 = c.b3;
b.b1 = 0;
b.b0 = c.b2;
```



Note

b0, b1, b2, and b3 represent the 8-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VEXT2BHM b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by zero-extending the higher two bytes of an operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = ((c >> 16) & 0x0000_00ff) | ((c >> 8) & 0x00ff_0000); /* VEXT2BHM*/
```

Assembly Code Example

```
VEXT2BHM r1,r2      ; zero extend two higher bytes from r2 into 16-bit
                    ; words in r1
```

Syntax and Encoding

		Instruction Code
VEXT2BHM	b, c	00101 bbb 00 101111 0 BBB CCCCC 100101
VEXT2BHM	b, limm	00101 bbb 00 101111 0 BBB 111110 100101
VEXT2BHM	0, c	00101 110 00 101111 0 111 CCCCC 100101
VEXT2BHM	0, limm	00101 110 00 101111 0 111 111110 100101
VEXT2BHM	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100101
VEXT2BHM	0, u6	00101 110 01 101111 0 111 uuuuuu 100101

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VEXT2BHMF

Function

Compose a vector with two Q15 elements created by expanding the higher two Q7 bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = c.b3;
b.b2 = 0;
b.b1 = c.b2;
b.b0 = 0;
```



Note

b0, b1, b2, and b3 represent the Q7 elements of an operand.

Instruction Format

op b,c

Syntax Example

VEXT2BHMF b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

MSB_aligned expansion of the higher two Q7 bytes of an operand into two Q15 elements This instruction does not modify any flags.

Pseudo Code

```
b = ((c & 0x00ff_0000)>>8) | ((c & 0xff00_0000)); /* VEXT2BHMF*/
```

Assembly Code Example

```
VEXT2BHMF r1,r2      ; Expand and pack 2 lower bytes from r1
                      ; into 2 16b words in r0
```

Syntax and Encoding

Instruction Code		
VEXT2BHMF	b, c	00101bbb001011110BBBCCCCC100001
VEXT2BHMF	b, u6	00101bbb011011110BBBuuuuuu100001

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VLSR2H

Function

Perform a two-way logical shift right operation on 16-bit vectors.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = unsigned (b.h1) >> c.h1;
a.h0 = unsigned (b.h0) >> c.h0;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VLSR2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Logic shift right the 16-bit vector elements of the operand *b* by the scalar *c*. The resulting vector is stored in the destination operand. The shift amount is unsigned and only the lower 4 bits of the vector shift amount is used. This instruction does not update any STATUS32 flags.

Pseudo Code

```

bl = b & 0x0000_ffff;
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_000f;
ch = (c >> 16) & 0x0000_000f;
al = bl >> cl;
ah = bh >> ch;
a = (al & 0x0000_ffff) | (ah << 16);
/* VLSR2H/ */

```

Assembly Code Example

```

VLSR2H r0,r1,r2      ; Logic shift right the operand r1 by scalar r2, and
                    ;return the result to r0.

```

Syntax and Encoding

Instruction Code

VLSR2H	a,b,c	00101bbb001000110BBBCCCCC	AAAAAA
VLSR2H	a,b,u6	00101bbb011000110BBBuuuuuu	AAAAAA
VLSR2H	b,b,s12	00101bbb101000110BBBssssss	SSSSSS
VLSR2H<.cc>	b,b,c	00101bbb111000110BBBCCCCC	0QQQQQ
VLSR2H<.cc>	b,b,u6	00101bbb111000110BBBuuuuuu	1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2H

Function

Signed multiplication and accumulation of two 16-bit vectors. For the non-DSP version, see [VMAC2H](#).

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
A.w1 = acchi+(b.h1 * c.h1);
A.w0 = acclo+(b.h0 * c.h0);
```



Note

`h0` and `h1` represent the 16-bit elements of a 32-bit operand.

Instruction Format

op A,b,c

Syntax Example

VMAC2H A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed integer multiplication of the higher 16 bits of `b` and `c` and add the product to the high accumulator. Similarly, perform a signed integer multiplication of the lower 16 bits of `b` and `c` and add the product to the low accumulator. Return the products as a two-way 32-bit vector in two consecutive cycles in a register pair `A`. `A` should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- Using the loop-counter (`LP_COUNT/r60`) as a destination operand register.
- Using an odd destination register for DSP instructions with 64-bit 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	00101bbb000111100BBBCCCCCAAAAAA
VMAC2H	a,b,u6	00101bbb010111100BBBuuuuuuAAAAAA
VMAC2H	b,b,s12	00101bbb100111100BBBssssssSSSSSS
VMAC2H<.cc>	b,b,c	00101bbb110111100BBBCCCCC0QQQQQ
VMAC2H<.cc>	b,b,u6	00101bbb110111100BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2HF

Function

Two-way signed multiplication and accumulation of two 16-bit fractional vectors. The result is saturated and stored.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16((acchi+(b.h1 * c.h1))<<1);
a.h0 = SAT16((acclo+=(b.h0 * c.h0))<<1);
```



Note

`h0` and `h1` represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMAC2HF a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and add the product to the high accumulator to get the result. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c` and add the product to the low accumulator to get the result. Return the results as a two-way 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMAC2HF*/
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        // guard bits are enabled
        acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(acclo, 31, &alo);
        acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(acchi, 31, &ahi);
    else
        // guard bits disabled
        sl = sat((b.h0 * c.h0 << 1), 31, &addlo);
        sh = sat((b.h1 * c.h1 << 1), 31, &addhi);
        acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(acclo, 31, &alo) || sl;
        acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(acchi, 31, &ahi) || sh;
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = 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;
    s1              = 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| = s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMAC2HF r0,r1,r2      ; Signed fractional multiply-accumulate two vectors
                       ;r1 and r2 and return result in r0

```

Syntax and Encoding

Instruction Code		
VMAC2HF	a,b,c	00101bbb000111101BBBCCCCCAAAAAA
VMAC2HF	a,b,u6	00101bbb010111101BBBuuuuuuAAAAAA
VMAC2HF	b,b,s12	00101bbb100111101BBBssssssSSSSSS
VMAC2HF<.cc>	b,b,c	00101bbb110111101BBBCCCCC0QQQQQ
VMAC2HF<.cc>	b,b,u6	00101bbb110111101BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2HFR

Function

Two-way signed multiplication and accumulation of two 16-bit fractional vectors. The result is rounded and saturated.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16 (RND16 ((acchi += (b.h1 * c.h1)) << 1));
a.h0 = SAT16 (RND16 ((acclo += (b.h0 * c.h0)) << 1));
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMAC2HFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of *b* and *c* and add the product to the high accumulator to get the result. Similarly, perform a signed fractional multiplication of the lower 16 bits of *b* and *c* and add the product to the low accumulator to get the result. Return the results as a two-way rounded 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMAC2HFR*/
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        sl = sat((b.h0 * c.h0 << 1), 31, &addlo);
        sh = sat((b.h1 * c.h1 << 1), 31, &addhi);
        acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo) || sl;
        acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi) || sh;
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMAC2HFR r0,r1,r2      ; Signed fractional multiply two vectors r1 and r2
                        ;and return result in r0 with saturation and rounding

```

Syntax and Encoding

Instruction Code		
VMAC2HFR	a,b,c	00101bbb000111111BBBCCCCCAAAAAA
VMAC2HFR	a,b,u6	00101bbb010111111BBBuuuuuuAAAAAA
VMAC2HFR	b,b,s12	00101bbb100111111BBBssssssSSSSSS
VMAC2HFR<.cc>	b,b,c	00101bbb110111111BBBCCCCC0QQQQQ
VMAC2HFR<.cc>	b,b,u6	00101bbb110111111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2HNFR

Function

Two-way signed multiplication and accumulation of two 16-bit fractional vectors. The result is rounded and saturated, and returned without a fractional shift.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16 (RND16 (acchi+ (b.h1 * c.h1))) ;
a.h0 = SAT16 (RND16 (acclo+ (b.h0 * c.h0))) ;
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMAC2HNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of *b* and *c* and add the product to the high accumulator to get the result. Similarly, perform a signed fractional multiplication of the lower 16 bits of *b* and *c* and add the product to the low accumulator to get the result. Return the results as a two-way rounded 16-bit fractional vector. The result is not shifted by one bit. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
// note: accumulators modeled as 128b integers                                     /* VMAC2HNFR*/
addlo = b.h0 * c.h0;
addhi = b.h1 * c.h1;
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(round(acclo,16), 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= sl | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);
```

Assembly Code Example

```
VMAC2HNFR r0,r1,r2      ; Signed fractional multiply two vectors r1 and r2
                        ;and return result in r0 with saturation and rounding
```

Syntax and Encoding

Instruction Code		
VMAC2HNFR	a,b,c	00110bbb000100010BBBCCCCCAAAAAA
VMAC2HNFR	a,b,u6	00110bbb010100010BBBuuuuuuAAAAAA

VMAC2HNFR	b, b, s12	00110 bbb 10 010001 0 BBB ssssss SSSSSS
VMAC2HNFR<.cc>	b, b, c	00110 bbb 11 010001 0 BBB CCCCC 0QQQQQ
VMAC2HNFR<.cc>	b, b, u6	00110 bbb 11 010001 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAC2HU

Function

Unsigned multiplication and accumulation of two 16-bit vectors. For the non-DSP version, see [VMAC2HU](#).

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
A.w1 = acchi+(b.h1 * c.h1);
A.w0 = acclo+(b.h0 * c.h0);
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op A,b,c

Syntax Example

VMAC2HU A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an unsigned integer multiplication of the higher 16 bits of *b* and *c* and add the product to the high accumulator. Similarly, perform an unsigned integer multiplication of the lower 16 bits of *b* and *c* and add the product to the low accumulator. Return the products as a two-way 32-bit vector in two consecutive cycles in 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	00101bbb000111110BBBCCCCCAAAAAA
VMAC2HU	a,b,u6	00101bbb010111110BBBuuuuuuAAAAAA
VMAC2HU	b,b,s12	00101bbb100111110BBBssssssSSSSSS
VMAC2HU<.cc>	b,b,c	00101bbb110111110BBBCCCCC0QQQQQ
VMAC2HU<.cc>	b,b,u6	00101bbb110111110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMAX2H

Function

Compare two-way 16-bit vectors and return the maximum.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = max (b.h1, c.h1);
a.h0 = max (b.h0, c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VMAX2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compare the 16-bit signed vector elements of the operand **b** and **c**. The maximum value of the two elements is stored in the destination operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

bl = (b & 0x0000_ffff) << 16;                                     /* VMAX2H/*
bh = ((b >> 16) & 0x0000_ffff) << 16;
cl = (c & 0x0000_ffff) << 16;
ch = ((c >> 16) & 0x0000_ffff) << 16;
al = bl >= cl ? bl : cl;
ah = bh >= ch ? bh : ch;
a = ((al >> 16) & 0x0000_ffff) | ah;

```

Assembly Code Example

```

VMAX2H r0,r1,r2          ; Compare the 16-bit vector elements of the operands
                          ;r2 and r1, and return the maximum to r0.

```

Syntax and Encoding

Instruction Code

VMAX2H	a,b,c	00101 bbb 00 100100 1 BBB CCCCC AAAAAA
VMAX2H	a,b,u6	00101 bbb 01 100100 1 BBB uuuuuu AAAAAA
VMAX2H	b,b,s12	00101 bbb 10 100100 1 BBB ssssss SSSSSS
VMAX2H<.cc>	b,b,c	00101 bbb 11 100100 1 BBB CCCCC 0QQQQQ
VMAX2H<.cc>	b,b,u6	00101 bbb 11 100100 1 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMIN2H

Function

Compare two-way 16-bit signed vectors and return the minimum.

Extension Group

DSP vector arithmetic

Operation

```
a.h1 = min (b.h1, c.h1);
a.h0 = min (b.h0, c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VMIN2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compare the 16-bit signed vector elements of the operand **b** and **c**. The minimum value of the two elements is stored in the destination operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

bl = (b & 0x0000_ffff) << 16;                                     /* VMIN2H/*
bh = ((b >> 16) & 0x0000_ffff) << 16;
cl = (c & 0x0000_ffff) << 16;
ch = ((c >> 16) & 0x0000_ffff) << 16;
al = bl < cl ? bl : cl;
ah = bh < ch ? bh : ch;
a = ((al >> 16) & 0x0000_ffff) | ah;

```

Assembly Code Example

```

VMIN2H r0,r1,r2          ; Compare the 16-bit vector elements of the operands
                          ;r2 and r1, and return the minimum to r0.

```

Syntax and Encoding

Instruction Code

VMIN2H	a,b,c	00101 bbb 00 1001011 BBBCCCCC AAAAAA
VMIN2H	a,b,u6	00101 bbb 01 1001011 BBBuuuuuu AAAAAA
VMIN2H	b,b,s12	00101 bbb 10 1001011 BBBssssss SSSSSS
VMIN2H<.cc>	b,b,c	00101 bbb 11 1001011 BBBCCCCC 0QQQQQ
VMIN2H<.cc>	b,b,u6	00101 bbb 11 1001011 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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2H

Function

Dual 16-bit SIMD multiplication. For the non-DSP version, see [VMPY2H](#).

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
if (cc) {
    acc.w0 = A.w0 = b.h0 * c.h0;
    acc.w1 = A.w1 = b.h1 * c.h1;
}
```

Instruction Format

op A, b, c

Syntax Example

VMPY2H A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed integer multiplication of the higher 16 bits of *b* and *c* and store the product to the high accumulator. Similarly, perform a signed integer multiplication of the lower 16 bits of *b* and *c* and store the product to the low accumulator. Return the products as a two-way 32-bit vector in two consecutive cycles in a register pair *A*. *A* should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit 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](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
VMPY2H	a,b,c	00101bbb0001111000BBBCCCCCAAAAAA
VMPY2H	a,b,u6	00101bbb0101111000BBBuuuuuuAAAAAA
VMPY2H	b,b,s12	00101bbb1001111000BBBssssssSSSSSS
VMPY2H<.cc>	b,b,c	00101bbb1101111000BBBCCCCC0QQQQQ
VMPY2H<.cc>	b,b,u6	00101bbb1101111000BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2HF

Function

Two-way signed multiplication of two 16-bit fractional vectors. The result is saturated and stored.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16((acchi = (b.h1 * c.h1)) << 1);
a.h0 = SAT16((acclo = (b.h0 * c.h0)) << 1);
```



Note

`h0` and `h1` represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMPY2HF a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and store the product to the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c` and store the product to the low accumulator. Return the products as a two-way 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* VMPY2HF*/
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        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) || 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(acchi, 31, &ahi) || sh;
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMPY2HF r0,r1,r2      ; Signed fractional multiply two vectors r1 and r2
                      ;and return result in r0

```

Syntax and Encoding

Instruction Code		
VMPY2HF	a,b,c	00101 bbb 00 011100 1 BBB CCCCC AAAAAA
VMPY2HF	a,b,u6	00101 bbb 01 011100 1 BBB uuuuuu AAAAAA
VMPY2HF	b,b,s12	00101 bbb 10 011100 1 BBB ssssss SSSSSS
VMPY2HF<.cc>	b,b,c	00101 bbb 11 011100 1 BBB CCCCC 0QQQQQ
VMPY2HF<.cc>	b,b,limm	00101 bbb 11 011100 1 BBB 111110 0QQQQQ

VMPY2HF<.cc> b,b,u6 00101**bbb**11**011100**1**BBB**uuuuuu**1QQQQQ**

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2HFR

Function

Two-way signed multiplication of two 16-bit fractional vectors. The result is saturated, rounded, and stored.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16(RND16((acchi = (b.h1 * c.h1))<<1));
a.h0 = SAT16(RND16((acclo = (b.h0 * c.h0))<<1));
```



Note

h0 and h1 represent the 16-bit elements of a 32-bit operand.

Instruction Format

op a,b,c

Syntax Example

VMPY2HFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of *b* and *c* and store the product to the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of *b* and *c* and store the product to the low accumulator. Return the products as a two-way rounded 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMPY2HFR*/
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = b.h0 * c.h0 << 1;
        addhi = b.h1 * c.h1 << 1;
        acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        sl = sat((b.h0 * c.h0 << 1), 31, &addlo);
        sh = sat((b.h1 * c.h1 << 1), 31, &addhi);
        acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo) || sl;
        acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi) || sh;
    else
        // post-accumulate fractional shift mode
        addlo = b.h0 * c.h0;
        addhi = b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(0, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(0, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(0, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(0, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMPY2HFR r0,r1,r2      ; Signed fractional multiply two vectors r1 and r2
                        ;and return the saturated and rounded result in r0

```

Syntax and Encoding

Instruction Code		
VMPY2HFR	a,b,c	00101 bbb 00 0111011 BBB CCCCC AAAAAA
VMPY2HFR	a,b,u6	00101 bbb 01 0111011 BBB uuuuuu AAAAAA
VMPY2HFR	b,b,s12	00101 bbb 10 0111011 BBB ssssss SSSSSS
VMPY2HFR<.cc>	b,b,c	00101 bbb 11 0111011 BBB CCCCC 0QQQQQ
VMPY2HFR<.cc>	b,b,u6	00101 bbb 11 0111011 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2HU

Function

Dual unsigned 16-bit SIMD multiplication. For the non-DSP version, see [VMPY2HU](#).

Extension Group

(MPY_OPTION > 7) or (HAS_DSP == 1)

Operation

```
if (cc) {
    acc.w0 = A.w0 = b.h0 * c.h0;
    acc.w1 = A.w1 = b.h1 * c.h1;
}
```

Instruction Format

op A,b, c

Syntax Example

VMPY2HU A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform an unsigned integer multiplication of the higher 16 bits of *b* and *c* and store the product to the high accumulator. Similarly, perform an unsigned integer multiplication of the lower 16 bits of *b* and *c* and store the product to the low accumulator. Return the products as a two-way 32-bit vector in two consecutive cycles in a register pair *A*. *A* should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (LP_COUNT/r60) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit 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](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
VMPY2HU	a,b,c	00101bbb000111010BBBCCCCCAAAAAA
VMPY2HU	a,b,u6	00101bbb010111010BBBuuuuuuAAAAAA
VMPY2HU	b,b,s12	00101bbb100111010BBBssssssSSSSSS
VMPY2HU<.cc>	b,b,c	00101bbb110111010BBBCCCCC0QQQQQ
VMPY2HU<.cc>	b,b,u6	00101bbb110111010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMPY2HWF

Function

Two-way signed multiplication of two 16-bit fractional vectors. A saturated 64-bit result is returned.

Extension Group

DSP Vector 16x16 MAC

Operation

```
A.w1 = SAT32 ( (acchi = (b.h1 * c.h1)) <<1 );
A.w0 = SAT32 ( (acchi = (b.h0 * c.h0)) <<1 );
```



Note

`h0` and `h1` represent the 16-bit elements of a 32-bit operand.

Instruction Format

op A,b,c

Syntax Example

VMPY2HWF A,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and store the product to the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c` and store the product to the low accumulator. Return the products as a two-way 32-bit fractional vector in a register pair `A`. `A` should be an even-numbered register. This instruction does not modify any flags.

An illegal instruction exception is raised in the following cases:

- ❑ Using the loop-counter (`LP_COUNT/r60`) as a destination operand register.
- ❑ Using an odd destination register for DSP instructions with 64-bit 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	00101bbb011000000BBBuuuuuuAAAAAA
VMPY2HWF	b,b,s12	00101bbb101000000BBBssssssSSSSSS
VMPY2HWF<.cc>	b,b,c	00101bbb111000000BBBCCCCC0QQQQQ
VMPY2HWF<.cc>	b,b,u6	00101bbb111000000BBBuuuuuu1QQQQQ

VMSUB2HF

Function

Two-way signed multiplication and subtraction of two 16-bit fractional vectors. The result is saturated and stored.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16((acchi -= (b.h1*c.h1)) <<1);
a.h0 = SAT16((acclo -= (b.h0*c.h0)) <<1);
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VMSUB2HF a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and subtract the product from the high accumulator and store the result in the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c`, subtract the product from the low accumulator, and store the result in the low accumulator. Return the results as a two-way 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                /* VMSUB2HF*/
if (DSP_CTRL.PA)
  // pre-accumulate fractional shift mode
  if (DSP_CTRL.GE)
    addlo = -b.h0 * c.h0 << 1;
    addhi = -b.h1 * c.h1 << 1;
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1
: ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo, 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1
: ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi, 31, &ahi);
  else
    // guard bits disabled
    sl = sat(-(b.h0 * c.h0 << 1), 31, &addlo);
    sh = sat(-(b.h1 * c.h1 << 1), 31, &addhi);
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo, 31, &alo) || sl;
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi, 31, &ahi) || sh;
  else
    // post-accumulate fractional shift mode
    addlo = -b.h0 * c.h0;
    addhi = -b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1
: ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1
: ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(acclo<<1, 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(acchi<<1, 31, &ahi);
DSP_CTRL.SAT |= sl | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMSUB2HF r1,r2,r3                ; Signed fractional multiply-
                                ;subtract two 16-bit vectors r2
                                ;and r3. return result in r1

```

Syntax and Encoding

Instruction Code

VMSUB2HF	a,b,c	00110 bbb 00 000100 0 BBB CCCCC AAAAAA
VMSUB2HF	a,b,u6	00110 bbb 01 000100 0 BBB uuuuuu AAAAAA
VMSUB2HF	b,b,s12	00110 bbb 10 000100 0 BBB ssssss SSSSSS
VMSUB2HF<.cc>	b,b,c	00110 bbb 11 000100 0 BBB CCCCC 0QQQQQ
VMSUB2HF<.cc>	b,b,u6	00110 bbb 11 000100 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMSUB2HFR

Function

Two-way signed multiplication and subtraction of two 16-bit fractional vectors. The result is rounded and saturated.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16(RND16((acchi -= (b.h1*c.h1))<<1));
a.h0 = SAT16(RND16((acclo -= (b.h0*c.h0))<<1));
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VMSUB2HFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and subtract the product from the high accumulator and store the result in the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c`, subtract the product from the low accumulator, and store the result in the low accumulator. Return the results as a two-way rounded 16-bit fractional vector. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

// note: accumulators modeled as 128b integers                                     /* VMSUB2HFR*/
if (DSP_CTRL.PA)
    // pre-accumulate fractional shift mode
    if (DSP_CTRL.GE)
        // guard bits are enabled
        addlo = -b.h0 * c.h0 << 1;
        addhi = -b.h1 * c.h1 << 1;
        acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
        ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo);
        acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
        ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi);
    else
        // guard bits disabled
        sl = sat(-(b.h0 * c.h0 << 1), 31, &addlo);
        sh = sat(-(b.h1 * c.h1 << 1), 31, &addhi);
        acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
        ACC0_GLO.G      = 0;
        ACC0_LO          = acclo & 0xffff_ffff;
        sl               = sat(round(acclo, 16), 31, &alo) || sl;
        acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
        ACC0_GHI.G      = 0;
        ACC0_HI          = acchi & 0xffff_ffff;
        sh               = sat(round(acchi, 16), 31, &ahi) || sh;
    else
        // post-accumulate fractional shift mode
        addlo = -b.h0 * c.h0;
        addhi = -b.h1 * c.h1;

```

```

if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
&ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    s1              = sat(round(acclo<<1, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
&ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi<<1, 16), 31, &ahi);
DSP_CTRL.SAT |= s1 | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);

```

Assembly Code Example

```

VMSUB2HFR r1,r2,r3          ; Signed fractional multiply-
                             ;subtract two 16-bit vectors r2 and
                             ;r3. return rounded result in r1

```

Syntax and Encoding

Instruction Code

VMSUB2HFR	a,b,c	00110 bbb 00 000011 0 BBB CCCCC AAAAAA
VMSUB2HFR	a,b,u6	00110 bbb 01 000011 0 BBB uuuuuu AAAAAA
VMSUB2HFR	b,b,s12	00110 bbb 10 000011 0 BBB ssssss SSSSSS
VMSUB2HFR<.cc>	b,b,c	00110 bbb 11 000011 0 BBB CCCCC 0QQQQQ
VMSUB2HFR<.cc>	b,b,u6	00110 bbb 11 000011 0 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element *g* or *h*, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VMSUB2HNFR

Function

Two-way signed multiplication and subtraction of two 16-bit fractional vectors. The result is saturated, and returned without a fractional shift.

Extension Group

DSP Vector 16x16 MAC

Operation

```
a.h1 = SAT16(RND16(acchi - (b.h1*c.h1)));
a.h0 = SAT16(RND16(acclo - (b.h0*c.h0)));
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VMSUB2HNFR a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Perform a signed fractional multiplication of the higher 16 bits of `b` and `c` and subtract the product from the high accumulator and store the result in the high accumulator. Similarly, perform a signed fractional multiplication of the lower 16 bits of `b` and `c`, subtract the product from the low accumulator, and store the result in the low accumulator. Return the results as a two-way rounded 16-bit fractional vector. The result is not shifted by one bit. This instruction does not modify any flags.

Pseudo Code

```
// note: accumulators modeled as 128b integers
addlo = -b.h0 * c.h0;
addhi = -b.h1 * c.h1;
if (DSP_CTRL.GE)
    // guard bits are enabled
    acclo = acc_add(acclo, addlo, 40, &ACC0_GLO.Z, &ACC0_GLO.N,
    &ACC0_GLO.V)
    ACC0_GLO.GUARD = (acclo >> 32) & 0xff;
    ACC0_GLO.G      = ((acclo >> 31) & 1) ? ACC0_GLO.GUARD != -1 :
ACC0_GLO.GUARD != 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 40, &ACC0_GHI.Z, &ACC0_GHI.N,
    &ACC0_GHI.V)
    ACC0_GHI.GUARD = (acchi >> 32) & 0xff;
    ACC0_GHI.G      = ((acchi >> 31) & 1) ? ACC0_GHI.GUARD != -1 :
ACC0_GHI.GUARD != 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi, 16), 31, &ahi);
else
    // guard bits are disabled
    acclo = acc_add(acclo, addlo, 32, &ACC0_GLO.Z, &ACC0_GLO.N,
    &ACC0_GLO.V)
    ACC0_GLO.G      = 0;
    ACC0_LO         = acclo & 0xffff_ffff;
    sl              = sat(round(acclo, 16), 31, &alo);
    acchi = acc_add(acchi, addhi, 32, &ACC0_GHI.Z, &ACC0_GHI.N,
    &ACC0_GHI.V)
    ACC0_GHI.G      = 0;
    ACC0_HI         = acchi & 0xffff_ffff;
    sh              = sat(round(acchi, 16), 31, &ahi);
DSP_CTRL.SAT |= sl | sh | ACC0_GLO.V | ACC0_GHI.V;
a = (ahi & 0xffff_0000) | ((alo >> 16) & 0x0000_ffff);
```

Assembly Code Example

```
VMSUB2HNFR r1,r2,r3                ; Signed fractional multiply-
                                   ;subtract two 16-bit vectors r2 and
                                   ;r3. return rounded result in r1
```

Syntax and Encoding

Instruction Code		
VMSUB2HNFR	a,b,c	00110bbb000100011BBBCCCCCAAAAAA
VMSUB2HNFR	a,b,u6	00110bbb010100011BBBuuuuuuAAAAAA

VMSUB2HNFR	b, b, s12	00110 bbb 10 010001 1 BBB ssssss SSSSSS
VMSUB2HNFR<.cc>	b, b, c	00110 bbb 11 010001 1 BBB CCCCC 0QQQQQ
VMSUB2HNFR<.cc>	b, b, u6	00110 bbb 11 010001 1 BBB uuuuuu 1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VNEG2H

Function

Negate the two 16-bit vectors of an operand and store the negated value in the destination operand.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = -c.h1;
b.h0 = -c.h0;
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VNEG2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Negate the two 16-bit vectors of an operand and store the negated value in the destination operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
b1 = 0 - c1;
bh = 0 - ch;
b = (b1 & 0x0000_ffff) | (bh << 16);
```

/* VNEG2H/*

Assembly Code Example

```
VNEG2H r1,r2          ; Negate vector r2 and return result in r1
```

Syntax and Encoding

Instruction Code		
VNEG2H	b, c	00101 bbb 00 101111 0 BBB CCCCC 101010
VNEG2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101010
VNEG2H	0, c	00101 110 00 101111 0 111 CCCCC 101010
VNEG2H	0, limm	00101 110 00 101111 0 111 111110 101010
VNEG2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101010
VNEG2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VNEGS2H

Function

Negate the two 16-bit vectors of an operand and store the saturated negated value in the destination operand.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = SAT16(-c.h1);
b.h0 = SAT16(-c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VNEGS2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Negate the two 16-bit vectors of an operand and store the saturated negated value in the destination operand. The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)0 - c1, 15, &b1);
sh = sat((sint128)0 - ch, 15, &bh);
DSP_CTRL.SAT |= s1 | sh;
b = (b1 & 0x0000_ffff) | (bh << 16);
/* VNEGS2H/*

```

Assembly Code Example

```
VNEGS2H r1,r2      ; Negate vector r2 and return saturated result in r1
```

Syntax and Encoding

Instruction Code

VNEGS2H	b, c	00101 bbb 00 101111 0 BBB CCCCC 101011
VNEGS2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101011
VNEGS2H	0, c	00101 110 00 101111 0 111 CCCCC 101011
VNEGS2H	0, limm	00101 110 00 101111 0 111 111110 101011
VNEGS2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101011
VNEGS2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101011

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VNORM2H

Function

Two-way 16-bit vector normalization.

Extension Group

DSP vector arithmetic

Operation

```
b.h1 = norm(c.h1);
b.h0 = norm(c.h0);
```



Note

h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VNORM2H b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Normalize a two-way 16-bit vector to return a vector of normalized constants. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

short normh(short c) {
    int i;
    for (i = 0; (i < 16) && ((c >> i) != 0) && ((c >> i) != -1); i++);
    return 15 - i;
}
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
b1 = normh(c1);
bh = normh(ch);
b = (b1 & 0x0000_ffff) | (bh << 16);

```

/*
VNORM2H/*

Assembly Code Example

```

VNORM2H r1,r2      ; Normalize each 16-bit vector in r2 and return
                   ;result in r1

```

Syntax and Encoding

Instruction Code

VNORM2H	b, c	00101 bbb 00 101110 0 BBB CCCCC 101100
VNORM2H	b, limm	00101 bbb 00 101111 0 BBB 111110 101100
VNORM2H	0, c	00101 110 00 101111 0 111 CCCCC 101100
VNORM2H	0, limm	00101 110 00 101111 0 111 111110 101100
VNORM2H	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 101100
VNORM2H	0, u6	00101 110 01 101111 0 111 uuuuuu 101100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HL

Function

Compose the destination operand from the lower 16-bits of the source operands.

Extension Group

DSP vector unpacking

Operation

```
a.h0 = c.h0;
a.h1 = b.h0;
```

Instruction Format

op a,b,c

Syntax Example

VPACK2HL a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose the destination operand from the lower 16-bits of the source operands. This instruction does not modify any flags.

Pseudo Code

```
a = ((b & 0x0000ffff) << 16) | (c & 0x0000ffff) /* VPACK2HL*/
```

Assembly Code Example

```
VPACK2HL r0,r1,r2 ; Pack lower 16 bits from r1 and r2 into r0
```

Syntax and Encoding

		Instruction Code
VPACK2HL	a, b, c	00101bbb001010010BBBCCCCCAAAAAA
VPACK2HL	a, b, u6	00101bbb011010010BBBuuuuuuAAAAAA
VPACK2HL	b, b, s12	00101bbb101010010BBBssssssSSSSSS
VPACK2HL<.cc>	b, b, c	00101bbb111010010BBBCCCCC0QQQQQ
VPACK2HL<.cc>	b, b, u6	00101bbb111010010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HM

Function

Compose the destination operand from the higher 16-bits of the source operands.

Extension Group

DSP vector unpacking

Operation

```
a.h0 = c.h1;
a.h1 = b.h1;
```

Instruction Format

op a,b,c

Syntax Example

VPACK2HM a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose the destination operand from the higher 16-bits of the source operands. This instruction does not modify any flags.

Pseudo Code

```
a = ((b & 0x0000ffff) | (c & 0x0000ffff) >> 16) /* VPACK2HM*/
```

Assembly Code Example

```
VPACK2HM r0,r1,r2 ; Pack higher 16 bits from r1 and r2 into r0
```

Syntax and Encoding

		Instruction Code
VPACK2HM	a, b, c	00101bbb001010011BBBCCCCCAAAAAA
VPACK2HM	a, b, u6	00101bbb011010011BBBuuuuuuAAAAAA
VPACK2HM	b, b, s12	00101bbb101010011BBBssssssSSSSSS
VPACK2HM<.cc>	b, b, c	00101bbb111010011BBBCCCCC0QQQQQ
VPACK2HM<.cc>	b, b, u6	00101bbb111010011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HBL

Function

Pack the LSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the lower 16 bits of the destination.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = 0;
b.b2 = 0;
b.b1 = c.b2;
b.b0 = c.b0;
```

Instruction Format

op b, c

Syntax Example

VPACK2HBL b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Extract the two least significant bytes from a dual packed 16-bit elements of an operand, and return the extracted bytes in the lower two bytes of the destination operand. This instruction does not modify any flags.

Pseudo Code

```
b = ((c & 0x00ff0000) >> 8) | (c & 0x000000ff) /* VPACK2HBL*/
```

Assembly Code Example

```
VPACK2HBL r0,r1      ; Pack two lower significant bytes from r1
                      ; into lower part of a word in r0
```

Syntax and Encoding

Instruction Code		
VPACK2HBL	b, c	00101bbb001011110BBBCCCCC011100
VPACK2HBL	b, u6	00101bbb011011110BBBuuuuuu011100

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands. Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified. For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HBLF

Function

Pack the MSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the lower 16 bits of the destination.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = 0;
b.b2 = 0;
b.b1 = c.b3;
b.b0 = c.b1;
```

Instruction Format

op b, c

Syntax Example

VPACK2HBLF b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Extract the two most significant bytes from a the two 16-bit elements of the source operand (c), and return the extracted bytes in the lower two bytes of the destination operand (b). This instruction does not modify any flags.

Pseudo Code

```
b = ((c & 0xff000000) >> 16) | ((c & 0x0000ff00) >> 8) /* VPACK2HBLF*/
```

Assembly Code Example

```
VPACK2HBLF r0,r1      ; Pack two most significant bytes from r1
                       ; into lower part of a word in r0
```

Syntax and Encoding

Instruction Code

VPACK2HBLF	b, c	00101 bbb 00 101111 0 BBB CCCCC 011110
VPACK2HBLF	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 011110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HBM

Function

Pack the LSBs of a 2x16 bit vector into a 2x8 bit vector. Result is stored in the higher 16 bits of the destination.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = c.b2;
b.b2 = c.b0;
b.b1 = 0;
b.b0 = 0;
```

Instruction Format

op b, c

Syntax Example

VPACK2HBM b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Extract the two most significant bytes from the 16-bit elements of the source operand (c), and return the extracted bytes in the higher two bytes of the destination operand (b). This instruction does not modify any flags.

Pseudo Code

```
b = ((c & 0x00ff0000) << 8) | ((c & 0x000000ff) << 16) /* VPACK2HBM*/
```

Assembly Code Example

```
VPACK2HBM r0,r1      ; Pack two lower significant bytes from r1
                      ; into upper part of a word in r0
```

Syntax and Encoding

Instruction Code		
VPACK2HBM	b, c	00101bbb001011110BBBCCCCC011101
VPACK2HBM	b, u6	00101bbb011011110BBBuuuuuu011101

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands. Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified. For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPACK2HBMF

Function

Pack the MSBs of a 2x16 bit vectors into a 2x8 bit vector. Result is stored in the higher 16 bits of the destination.

Extension Group

DSP vector unpacking

Operation

```
b.b3 = c.b3;
b.b2 = c.b1;
b.b1 = 0;
b.b0 = 0;
```

Instruction Format

op b, c

Syntax Example

VPACK2HBMF b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Extract the two most significant bytes from a dual packed 16b half-word (c). Return the two bytes in the upper two bytes of operand b. This instruction does not modify any flags.

Pseudo Code

```
b = (c & 0xff000000) | ((c & 0x0000ff00) << 8) /* VPACK2HBMF*/
```

Assembly Code Example

```
VPACK2HBMF r0,r1    ; Pack two most significant bytes from r1
                    ; into upper part of a word in r0
```

Syntax and Encoding

Instruction Code

VPACK2HBMF	b, c	00101 bbb 00 101111 0 BBB CCCCC 011111
VPACK2HBMF	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 011111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VPERM

Function

Perform byte permutation with zero or sign extension.

Extension Group

DSP vector unpacking

Operation

$$a = \text{vperm}(b, c)$$

Instruction Format

$$\text{op } a, b, c$$

Syntax Example

$$\text{VPERM } a, b, c$$

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Swap bytes in a word and optionally zero or sign extend a byte.

Pseudo Code

```
// get three bit byte index value                                /* VPERM*/
index[0] = (c>>0) & 0x7;
index[1] = (c>>3) & 0x7;
index[2] = (c>>6) & 0x7;
index[3] = (c>>9) & 0x7;
// get one bit extend enable per byte
ext[0] = (c >> 12) & 1
ext[1] = (c >> 13) & 1
ext[2] = (c >> 14) & 1
ext[3] = (c >> 15) & 1
// do for each byte
sgn = 0;
a = 0;
for (i = 0; < 4; i++) {
    if (ext[i]) {
        // extend
        if ((index[i] & 1) == 0) {
            // zero extend
            byte = 0;
        } else {
            // sign extend
            byte = sgn ? 0xFF : 0x00;
        }
    } else if (index[i] < 4) {
        // copy byte from location index[i]
        byte = (b >> (8*index[i])) & 0xff;
    } else {
        // byte is out of range
        byte = 0
    }
    a = a | (byte << (8*i)); // add byte to output
    sgn = (byte & 0x80) != 0; // store sign for next
iteration
}
```

Assembly Code Example

```
VPERM r0,r1,r2          ; Permute the bytes from r2 as defined in r1
                        ; and writeback the result in r0
```

Syntax and Encoding

Instruction Code		
VPERM	a,b,c	00101bbb001011100BBBCCCCCAAAAAA
VPERM	a,b,u6	00101bbb011011100BBBuuuuuuAAAAAA

VPERM	b, b, s12	00101 bbb 10 101110 0 BBB ssssss SSSSSS
VPERM<.cc>	b, b, c	00101 bbb 11 101110 0 BBB CCCCC0 QQQQQ
VPERM<.cc>	b, b, u6	00101 bbb 11 101110 0 BBB uuuuu1 QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VREP2HL

Function

Compose a dual 16-bit vector by replicating the least-significant 16 bits of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.h1 = c.h0;
b.h0 = c.h0;
```



Note

`h0` and `h1` represent the 16-bit elements of an operand.

Instruction Format

op b, c

Syntax Example

VREP2HL b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by replicating the least-significant 16 bits of the `c` operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b = (c << 16) | (c & 0x0000_ffff); /* VREP2HL*/
```

Assembly Code Example

```
VREP2HL r1,r2      ; replicate lower 16-bits of r2 into r1.
```

Syntax and Encoding

Instruction Code		
VREP2HL	b, c	00101 bbb 00 101111 0 BBB CCCCC 100010
VREP2HL	b, limm	00101 bbb 00 101111 0 BBB 111110 100010
VREP2HL	0, c	00101 110 00 101111 0 111 CCCCC 100010
VREP2HL	0, limm	00101 110 00 101111 0 111 111110 100010
VREP2HL	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100010
VREP2HL	0, u6	00101 110 01 101111 0 111 uuuuuu 100010

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VREP2HM

Function

Compose a dual 16-bit vector by replicating the most-significant 16 bits of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.h1 = c.h1;
b.h0 = c.h1;
```



Note

`h0` and `h1` represent the 16-bit elements of an operand.

Instruction Format

op b, c

Syntax Example

VREP2HM b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by replicating the most-significant 16 bits of the `c` operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b = (c & 0xffff_0000) | ((c >>16) & 0x0000_ffff);    /* VREP2HM*/
```


Assembly Code Example

```
VREP2HM r1,r2      ; replicate higher 16-bits of r2 into r1.
```

Syntax and Encoding

Instruction Code		
VREP2HM	b, c	00101 bbb 00 101111 0 BBB CCCCC 100011
VREP2HM	b, limm	00101 bbb 00 101111 0 BBB 111110 100011
VREP2HM	0, c	00101 110 00 101111 0 111 CCCCC 100011
VREP2HM	0, limm	00101 110 00 101111 0 111 111110 100011
VREP2HM	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100011
VREP2HM	0, u6	00101 110 01 101111 0 111 uuuuuu 100011

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSEXT2BHL

Function

Compose a dual 16-bit vector by sign-extending the lower two bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.h1 = sext(c.b1);
b.h0 = sext(c.b0);
```



Note

b0 and b1 represent the 8-bit elements of an operand. h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b, c

Syntax Example

VSEXT2BHL a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by sign-extending the lower two bytes of the c operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = ((c << 24) >> 24) & 0x0000_ffff) | (((c & 0x0000_ff00) << 16) >> 8); /* VSEXT2BHL*/
```

Assembly Code Example

```
VSEXT2BHL r1,r2      ; sign extend two lower bytes from r2 into 16-bit
                      ; words in r1
```

Syntax and Encoding

Instruction Code		
VSEXT2BHL	b, c	00101 bbb 00 101111 0 BBB CCCCC 100110
VSEXT2BHL	b, limm	00101 bbb 00 101111 0 BBB 111110 100110
VSEXT2BHL	0, c	00101 110 00 101111 0 111 CCCCC 100110
VSEXT2BHL	0, limm	00101 110 00 101111 0 111 111110 100110
VSEXT2BHL	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100110
VSEXT2BHL	0, u6	00101 110 01 101111 0 111 uuuuuu 100110

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSEXT2BHM

Function

Compose a dual 16-bit vector by sign-extending the higher two bytes of an operand.

Extension Group

DSP vector unpacking

Operation

```
b.h1 = sext(c.b3);
b.h0 = sext(c.b2);
```



Note

b2 and b3 represent the 8-bit elements of an operand. h0 and h1 represent the 16-bit elements of an operand.

Instruction Format

op b,c

Syntax Example

VSEXT2BHM b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Compose a dual 16-bit vector by sign-extending the higher two bytes of the c operand. This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```
b = ((c << 8) >> 24) & 0x0000_ffff | ((c & 0xff00_0000) >> 8); /* VSEXT2BHM*/
```

Assembly Code Example

```
VSEXT2BHM r1,r2      ; sign extend two higher bytes from r2 into 16-bit
                      ; words in r1
```

Syntax and Encoding

Instruction Code		
VSEXT2BHM	b, c	00101 bbb 00 101111 0 BBB CCCCC 100111
VSEXT2BHM	b, limm	00101 bbb 00 101111 0 BBB 111110 100111
VSEXT2BHM	0, c	00101 110 00 101111 0 111 CCCCC 100111
VSEXT2BHM	0, limm	00101 110 00 101111 0 111 111110 100111
VSEXT2BHM	b, u6	00101 bbb 01 101111 0 BBB uuuuuu 100111
VSEXT2BHM	0, u6	00101 110 01 101111 0 111 uuuuuu 100111

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUB2

Function

Dual 32-bit SIMD subtract. For the non-DSP version, see [VSUB2](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.w0 = B.w0 - C.w0;
    A.w1 = B.w1 - C.w1;
}
```



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”](#).

Instruction Format

op A,B,C

Syntax Example

VSUB2 A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing two 32-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each pair of elements from B and C is subtracted to form two 32-bit differences. These differences are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {                                     /* VSUB2*/
    A.w0 = B.w0 - C.w0;
    A.w1 = B.w1 - C.w1;}
```

Assembly Code Example

```
VSUB2 r0,r2,r4                               ; dual SIMD 32-bit subtraction of
                                              ; (r3,r2) and (r5,r4). The result
                                              ; is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
VSUB2	a,b,c	00101bbb001111010BBBCCCCCAAAAAA
VSUB2	a,b,u6	00101bbb011111010BBBuuuuuuAAAAAA
VSUB2	b,b,s12	00101bbb101111010BBBssssssSSSSSS
VSUB2<.cc>	b,b,c	00101bbb111111010BBBCCCCC0QQQQQ
VSUB2<.cc>	b,b,u6	00101bbb111111010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUB2H

Function

Dual 16-bit vector subtraction. For the non-DSP version, see [VSUB2H](#).

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Each pair of elements from b and c is subtracted to form two 16-bit differences. These differences are assigned to the destination register, if defined.

This instruction does not modify any flags. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if(cc) {                                     /* VSUB2H*/
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 - c.h1;
}
```

Assembly Code Example

```
VSUB2H r1,r2,r3                ; dual SIMD 16-bit subtraction
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUB2H	a, b, c	00101 bbb 00 0101010 BBBCCCCC AAAAAA
VSUB2H	a, b, u6	00101 bbb 01 0101010 BBBuuuuuu AAAAAA
VSUB2H	b, b, s12	00101 bbb 10 0101010 BBBssssss SSSSSS
VSUB2H<.cc>	b, b, c	00101 bbb 11 0101010 BBBCCCCC 0QQQQQ
VSUB2H<.cc>	b, b, u6	00101 bbb 11 0101010 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:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUB4H

Function

Quad 16-bit SIMD subtraction. For the non-DSP version, see [VSUB4H](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.h0 = B.h0 - C.h0;
    A.h1 = B.h1 - C.h1;
    A.h2 = B.h2 - C.h2;
    A.h3 = B.h3 - C.h3;
}
```



Note

h0, h1, h2, and h3 represent the 16-bit elements of a 32-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"](#).

Instruction Format

op A,B,C

Syntax Example

VSUB4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each pair of elements from B and C is subtracted to form four 16-bit differences. These differences are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {      A.h0 = B.h0 - C.h0;          /* VSUB4H*/
    A.h1 = B.h1 - C.h1;
    A.h2 = B.h2 - C.h2;
    A.h3 = B.h3 - C.h3;
}
```

Assembly Code Example

```
VSUB4H r0,r2,r4          ; quad SIMD 16-bit subtraction
                          ;(r3,r2) and (r5,r4). The result
                          ;is stored in (r1,r0)
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

For more information about the encodings, see “[Key for 32-bit Addressing Modes and Encoding Conventions](#)” and “[Key for 16-bit Addressing Modes and Encoding Conventions](#)”.

Instruction Code		
VSUB4H	a,b,c	00101bbb001110010BBBCCCCCAAAAAA
VSUB4H	a,b,u6	00101bbb011110010BBBuuuuuuAAAAAA
VSUB4H	b,b,s12	00101bbb101110010BBBsssssssSSSSSS
VSUB4H<.cc>	b,b,c	00101bbb111110010BBBCCCCC0QQQQQ
VSUB4H<.cc>	b,b,u6	00101bbb111110010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBADD

Function

Dual 32-bit SIMD subtract and add. For the non-DSP version, see [VSUBADD](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.w0 = B.w0 - C.w0;
    A.w1 = B.w1 + C.w1;
}
```



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"](#).

Instruction Format

op A,B,C

Syntax Example

VSUBADD A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing two 32-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. The first pair of elements from B and C is subtracted to form a 32-bit difference. The second pair of elements is added to form a 32-bit sum. The difference and sum are assigned to the destination register pair, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if(cc) {                                     /* VSUBADD*/
    A.w0 = B.w0 - C.w0;
    A.w1 = B.w1 + C.w1;
}
```

Assembly Code Example

```
VSUBADD r0,r2,r4    ; dual SIMD 32-bit subtraction ;and addition of (r3,r2)
                   ;and (r5,r4). The result is stored in (r1,r0)
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUBADD	a, b, c	00101bbb001111110BBBCCCCCAAAAAA
VSUBADD	a, b, u6	00101bbb011111110BBBuuuuuuAAAAAA
VSUBADD	b, b, s12	00101bbb101111110BBBssssssSSSSSS
VSUBADD<.cc>	b, b, c	00101bbb111111110BBBCCCCC0QQQQQ
VSUBADD<.cc>	b, b, u6	00101bbb111111110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBADD2H

Function

Dual 16-bit vector subtraction and addition. For the non-DSP version, see [VSUBADD2H](#).

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 + c.h1;
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUBADD2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 32-bit *b* and *c* operands specify vectors containing two 16-bit elements. The lower 16-bit elements from *b* and *c* are subtracted to form a 16-bit difference. The higher 16-bit elements are added to form a 16-bit sum. The difference and sum are assigned to the destination register.

This instruction does not modify any flags. An illegal instruction exception is raised if LP_COUNT (r60) is given as the destination register.

Pseudo Code

```
if(cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 + c.h1;
}
```

/* VSUBADD2H*/

Assembly Code Example

```
VSUBADD2H r1,r2,r3    ; dual SIMD 16-bit subtraction and addition of r2
                      ; and r3
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUBADD2H	a, b, c	00101bbb000101110BBBCCCCCAAAAA
VSUBADD2H	a, b, u6	00101bbb010101110BBBuuuuuuAAAAA
VSUBADD2H	b, b, s12	00101bbb100101110BBBssssssSSSSSS
VSUBADD2H<.cc>	b, b, c	00101bbb110101110BBBCCCCC0QQQQQ
VSUBADD2H<.cc>	b, b, u6	00101bbb110101110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBADD4H

Function

Quad16-bit SIMD subtract and add. For the non-DSP version, see [VSUBADD4H](#).

Extension Group

MPY_OPTION == 9 or has_dsp==true

Operation

```
if (cc) {
    A.h0 = B.h0 - C.h0;
    A.h1 = B.h1 + C.h1;
    A.h2 = B.h2 - C.h2;
    A.h3 = B.h3 + C.h3;
}
```



Note

$h0$, $h1$, $h2$, and $h3$ represent the 16-bit elements of a 32-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"](#).

Instruction Format

op a, b, c

Syntax Example

VSUBADD4H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. These are normally register pairs, although the operand formats are not restricted to specifying registers. Each odd-numbered pair of elements from B and C is added to form 16-bit sums. Each even-numbered pair of elements from B and C is subtracted to form 16-bit differences. The four elemental results are assigned to the destination register, if defined.

This instruction does not modify any flags.

Pseudo Code

```
if (cc) ==true then                                /* VSUBADD4H*/
    A.h0 = B.h0 - C.h0
    A.h1 = B.h1 + C.h1
    A.h2 = B.h2 - C.h2
    A.h3 = B.h3 + C.h3
```

Assembly Code Example

```
VSUBADD4H r0,r2,r4    ; quad SIMD 16-bit subtraction and addition of r2
                      ;and r4
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUBADD4H	a, b, c	00101bbb001110110BBBCCCCCAAAAAA
VSUBADD4H	a, b, u6	00101bbb011110110BBBuuuuuuAAAAAA
VSUBADD4H	b, b, s12	00101bbb101110110BBBssssssSSSSSS
VSUBADD4H<.cc>	b, b, c	00101bbb111110110BBBCCCCC0QQQQQ
VSUBADD4H<.cc>	b, b, u6	00101bbb111110110BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUB2H

Function

Dual 16-bit vector subtraction. For the non-DSP version, see [VSUB2H](#).

Extension Group

(MPY_OPTION > 6) or (HAS_DSP == 1)

Operation

```
if (cc) {
    a.h0 = b.h0 - c.h0;
    a.h1 = b.h1 - c.h1;
}
```



Note

`h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUB2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Each pair of elements from `b` and `c` is subtracted to form two 16-bit differences. These differences are assigned to the destination register, if defined.

This instruction does not modify any flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b1 = b & 0x0000_ffff; /* VSUB2H*/
bh = (b >> 16) & 0x0000_ffff;
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
a1 = b1 - c1;
ah = bh - ch;
a = (a1 & 0x0000_ffff) | (ah << 16);
```

Assembly Code Example

```
VSUB2H r1,r2,r3 ; dual SIMD 16-bit subtraction
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
VSUB2H	a,b,c	00101bbb000101010BBBCCCCCAAAAAA
VSUB2H	a,b,u6	00101bbb010101010BBBuuuuuuAAAAAA
VSUB2H	b,b,s12	00101bbb100101010BBBssssssSSSSSS
VSUB2H<.cc>	b,b,c	00101bbb110101010BBBCCCCC0QQQQQ
VSUB2H<.cc>	b,b,u6	00101bbb110101010BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBS2

Function

Two-way 32-bit vector subtraction. The result is saturated.

Extension Group

dsp_wide == true

Operation

```
if (cc) {
    A.w0 = SAT32 (B.w0 - C.w0);
    A.w1 = SAT32 (B.w1 - C.w1);
}
```

Instruction Format

op a, b, c

Syntax Example

VSUBS2 A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The B and C operands are 64-bit operands. Each pair of elements from B and C is subtracted to form two 32-bit differences. These saturated differences are assigned to destination register.

Pseudo Code

```
s = sat((sint128) b.w0 - c.w0, 31, &a.w0) | /* VSUBS2*/
sat((sint128) b.w1 - c.w1, 31, &a.w1);
DSP_CTRL.SAT |= s;
```

Assembly Code Example

```
VSUBS2 r0,r2,r4      ; Subtract {r2,r3} and {r4,r5} and write result
                     ; into {r0,r1} while saturating
```

Syntax and Encoding

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
VSUBS2	a, b, c	00101bbb001111011BBBCCCCCAAAAAA
VSUBS2	a, b, u6	00101bbb011111011BBBuuuuuuAAAAAA
VSUBS2	b, b, s12	00101bbb101111011BBBssssssSSSSSS
VSUBS2<.cc>	b, b, c	00101bbb111111011BBBCCCCC0QQQQQ
VSUBS2<.cc>	b, b, u6	00101bbb111111011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBS2H

Function

Dual 16-bit vector subtraction. The result is saturated.

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {
    a.h0 = SAT16 (b.h0 - c.h0);
    a.h1 = SAT16 (b.h1 - c.h1);
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUBS2H a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Each pair of elements from b and c is subtracted to form two 16-bit differences. These saturated differences are assigned to destination register.

The saturation flag, `DSP_CTRL.SAT` is set if any of the two results of the instruction saturate regardless of the `.F` suffix. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```

b1 = b & 0x0000_ffff;                                /* VSUBS2H*/
bh = (b >> 16) & 0x0000_ffff;
c1 = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)b1 - c1, 15, a1);
sh = sat((sint128)bh - ch, 15, ah);
a = (a1 & 0x0000_ffff) | (ah << 16);
DSP_CTRL.SAT |= s

```

Assembly Code Example

```

VSUBS2H r1,r2,r3                                ; dual SIMD 16-bit subtraction
                                                ;and store the saturated result

```

Syntax and Encoding

Instruction Code

VSUBS2H	a,b,c	00101 bbb 00 0101011 BBBCCCCC AAAAAA
VSUBS2H	a,b,u6	00101 bbb 01 0101011 BBBuuuuuu AAAAAA
VSUBS2H	b,b,s12	00101 bbb 10 0101011 BBBssssss SSSSSS
VSUBS2H<.cc>	b,b,c	00101 bbb 11 0101011 BBBCCCCC 0QQQQQ
VSUBS2H<.cc>	b,b,u6	00101 bbb 11 0101011 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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBS4H

Function

Four-way 16-bit vector subtraction. The result is saturated.

Extension Group

dsp_wide==true

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”](#).

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” and “Key for 16-bit Addressing Modes and Encoding Conventions”.

Instruction Code		
VSUBS4H	a,b,c	00101bbb0011110011BBBCCCCCAAAAAA
VSUBS4H	a,b,u6	00101bbb011110011BBBuuuuuuAAAAAA
VSUBS4H	b,b,s12	00101bbb1011110011BBBssssssSSSSSS
VSUBS4H<.cc>	b,b,c	00101bbb111110011BBBCCCCC0QQQQQ
VSUBS4H<.cc>	b,b,u6	00101bbb111110011BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUB4B

Function

Four-way eight-bit vector subtraction.

Extension Group

DSP vector arithmetic

Operation

$$\begin{aligned} a.b3 &= b.b3 - c.b3; \\ a.b2 &= b.b2 - c.b2; \\ a.b1 &= b.b1 - c.b1; \\ a.b0 &= b.b0 - c.b0; \end{aligned}$$


Note

$b0$, $b1$, $b2$, and $b3$ represent the 8-bit elements of an operand.

Instruction Format

op a,b,c

Syntax Example

VSUB4B a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

Pair-wise byte subtraction of the b and c operands, and the results are stored in the a operand. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (LP_COUNT/r60) as a destination operand register.

Pseudo Code

```

b1l = b & 0x0000_00ff;
b1h = b & 0x0000_ff00;
bhl = b & 0x00ff_0000;
bhh = b & 0xff00_0000;
c1l = c & 0x0000_00ff;
c1h = c & 0x0000_ff00;
chl = c & 0x00ff_0000;
chh = c & 0xff00_0000;
all = b1l - c1l;
alh = b1h - c1h;
ahl = bhl - chl;
ahh = bhh - chh;
a = (all & 0x0000_00ff) | (alh & 0x0000_ff00) | (ahl & 0x00ff_0000) |
    ahh;

```

/* VSUB4B/*

Assembly Code Example

```

VSUB4B r0,r1,r2      ; Subtract two vectors r1 and r2, and store the
                      result in ;r0;

```

Syntax and Encoding

Instruction Code

VSUB4B	a,b,c	00101 bbb 00 1001010 BBBCCCCC AAAAAA
VSUB4B	a,b,u6	00101 bbb 01 1001010 BBBuuuuuu AAAAAA
VSUB4B	b,b,s12	00101 bbb 10 1001010 BBBssssss SSSSSS
VSUB4B<.cc>	b,b,c	00101 bbb 11 1001010 BBBCCCCC 0QQQQQ
VSUB4B<.cc>	b,b,u6	00101 bbb 11 1001010 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:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBADDS

Function

Two-way 32-bit vector subtraction and addition. The results are saturated.

Extension Group

dsp_wide==true

Operation

```
if (cc) {
    a.w0 = SAT32(b.w0 - c.w0);
    a.w1 = SAT32(b.w1 + c.w1);
}
```

Instruction Format

op a, b, c

Syntax Example

VSUBADDS a,b,c

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The *b* and *c* operands are 64-bit operands. The lower 32-bit elements from *b* and *c* are subtracted to form a 32-bit difference. The higher 32-bit elements are added to form a 32-bit sum. The difference and sum are assigned to the destination register.

Pseudo Code

```
s = sat((sint128) b.w0 - c.w0, 31, &a.w0) | /* VSUBADDS*/
sat((sint128) b.w1 + c.w1, 31, &a.w1);
DSP_CTRL.SAT |= s;
```

Assembly Code Example

```
VSUBADDS r0,r2,r4      ; Add/subtract {r2,r3} and {r4,r5} and write result
                        ; into {r0,r1} while saturating
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code		
VSUBADDS	a, b, c	00101bbb001111111BBBCCCCCAAAAAA
VSUBADDS	a, b, u6	00101bbb011111111BBBuuuuuuAAAAAA
VSUBADDS	b, b, s12	00101bbb101111111BBBssssssSSSSSS
VSUBADDS<.cc>	b, b, c	00101bbb111111111BBBCCCCC0QQQQQ
VSUBADDS<.cc>	b, b, u6	00101bbb111111111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBADDS2H

Function

Dual 16-bit vector subtraction and addition. The results are saturated.

Extension Group

DSP vector arithmetic

Operation

```
if (cc) {  
    a.h0 = SAT16 (b.h0 - c.h0);  
    a.h1 = SAT16 (b.h1 + c.h1);  
}
```



Note

h0 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUBADDS2H a,b,c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

The 32-bit `b` and `c` operands specify vectors containing two 16-bit elements. The lower 16-bit elements from `b` and `c` are subtracted to form a 16-bit difference. The higher 16-bit elements are added to form a 16-bit sum. The difference and sum are assigned to the destination register.

The saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix if the results of this instruction saturate. This instruction does not update any STATUS32 flags. An illegal instruction exception is raised if you use the loop-counter (`LP_COUNT/r60`) as a destination operand register.

Pseudo Code

```
b1 = b & 0x0000_ffff; /* VSUBADDS2H*/
bh = (b >> 16) & 0x0000_ffff;
cl = c & 0x0000_ffff;
ch = (c >> 16) & 0x0000_ffff;
s1 = sat((sint128)b1 - cl, 15, al);
sh = sat((sint128)bh + ch, 15, ah);
a = (al & 0x0000_ffff) | (ah << 16);
DSP_CTRL.SAT |= s
```

Assembly Code Example

```
VSUBADDS2H r1,r2,r3 ; dual SIMD 16-bit subtraction and addition and
                   ;store the saturated result
```

Syntax and Encoding

 **Note**

For detailed information about vector operands, see “[Vector Operands](#)”. For detailed information about expansion of literals, see “[Expansion of Literals in Vector Instructions](#)” and “[Expansion of Literals](#)”.

Instruction Code		
VSUBADDS2H	a,b,c	00101bbb000101111BBBCCCCCAAAAAA
VSUBADDS2H	a,b,u6	00101bbb010101111BBBuuuuuuAAAAAA
VSUBADDS2H	b,b,s12	00101bbb100101111BBBssssssSSSSSS
VSUBADDS2H<.cc>	b,b,c	00101bbb110101111BBBCCCCC0QQQQQ
VSUBADDS2H<.cc>	b,b,u6	00101bbb110101111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand. A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0,R0,R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

VSUBADDS4H

Function

Four-way 16-bit vector subtraction and addition. The results are saturated.

Extension Group

dsp_wide==true

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 (lower) and h1 (upper) represent the 16-bit elements of a 32-bit operand. For more information about vector elements, see [“Data Formats”](#).

Instruction Format

op a, b, c

Syntax Example

VSUBADDS4H A,B,C

STATUS32 Flags Affected

Z	☐	= Unchanged
N	☐	= Unchanged
C	☐	= Unchanged
V	☐	= Unchanged

Description

The 64-bit B and C operands specify vectors containing four 16-bit elements. Each operand contains 16-bit elements (h0 through h3). The h0 and h2 elements of the C operand are subtracted from the respective elements of the B operand, and the saturated result is stored in the respective elements of the destination operand. The h1 and h3 elements of the C operand are added to the respective elements of the B operand and the saturated result is stored in the respective elements of the destination operand.

The saturation flag, `DSP_CTRL.SAT`, is set regardless of the `.F` suffix if any of the results of this instruction saturate.

Pseudo Code

```
s = sat((sint128)b.h0 - c.h0, 15, &a.h0) | /* VSUBADDS4H*/
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

```
VSUBADDS4H r0,r2,r4 ; Subtract/add four vectors {r2,r3} and {r4,r5} and
; return saturated result in {r0,r1}
```

Syntax and Encoding



Note

For detailed information about vector operands, see [“Vector Operands”](#). For detailed information about expansion of literals, see [“Expansion of Literals in Vector Instructions”](#) and [“Expansion of Literals”](#).

For more information about the encodings, see [“Key for 32-bit Addressing Modes and Encoding Conventions”](#) and [“Key for 16-bit Addressing Modes and Encoding Conventions”](#).

Instruction Code

VSUBADDS4H	a, b, c	00101bbb001110111BBBCCCCCAAAAAA
VSUBADDS4H	a, b, u6	00101bbb011110111BBBuuuuuuAAAAAA
VSUBADDS4H	b, b, s12	00101bbb101110111BBBsssssssSSSSSS
VSUBADDS4H<.cc>	b, b, c	00101bbb111110111BBBCCCCC0QQQQQ
VSUBADDS4H<.cc>	b, b, u6	00101bbb111110111BBBuuuuuu1QQQQQ

Long Immediate and Null Destination Operands

In 32-bit formats, a register encoding of 62 represents a long-immediate (limm) source operand and a null destination operand.

A null destination operand is specified syntactically using a zero destination. For example, the following instruction subtracts r1 from r0, sets the flags, and then discards its result because it has a null destination register operand:

```
SUB.F 0, R0, R1
```

Two limm source operands in the same instruction must always have the same value. This is due to the fact that each instruction can have at most one limm value, although it can be referenced by both source operands.

Where permitted in 16-bit formats, a register encoding of 30 represents a long-immediate (limm) source operand and a null (0) destination operand. Most register operands in 16-bit formats are 3 bits wide, and are unable to specify a register encoding of 30. However, some 16-bit formats contain an extended 5-bit field, indicated with the syntactic element g or h, allowing a value of 30 to be specified.

For additional encoding and syntax notes for multiple limm values, see section Dual-operand Instructions, F32_GEN4.

Part 5

Appendices

In this part:

- [ISA Version Differences](#)
- [Implementation-dependent Behavior](#)
- [ARConnect Commands Details](#)

A

ISA Version Differences

This appendix summarizes the differences between ARCV2, ARCompact for ARC 600, and ARCompact for ARC 700.

The following are the most significant differences:

- Major opcode 0x08, previously listed in the ARCompact PRM as “User 32-bit extension instructions,” is reassigned to encode 16-bit ARC extensions.
- Major opcodes 0x09, 0x0A, and 0x0B, previously listed in the ARCompact PRM as “ARC market-specific extension instructions (32-bit),” are assigned in ARCV2 as 16-bit extension formats.
- The format of major opcode 0x0E has been revised to permit twice as many operators to be encoded, at the cost of reducing the width of the ‘h’ register operand from 6 to 5 bits. Instructions from ARCompact that used this format are not binary compatible with ARCV2, and *vice versa*.
- The major opcode 0x0F, [0x00-0x1F], General Register Format, now has a re-assignment of sub-opcode 0x0C, from MUL_S to MPY_S. When MPY_OPTION is set to include 32-bit multiply instructions, this encoding is used as a synonym for the 32-bit signed MPY instruction. When MPY_OPTION defines that either no multiply instructions are present or only 16x16 multiply operations are present, any use of sub-opcode 0x0C raises an Instruction Error exception.
- The J.F [ILINKn] instruction is not supported in ARCV2. The RTIE instruction handles all interrupt and exception returns. Use of the .F notation in a J instruction is silently ignored because this bit is reserved in these formats.
- The ARCV2 architecture has only one trap instruction. The TRAP0 instruction of the V1-700 architecture is replaced by the SWI instruction found in the V1-600 architecture.
- The ARCV2 architecture alters the semantics of SWI compared with the V1-700 TRAP0/SWI (which one could argue is not defined correctly). The ARCV2 architecture does not commit the SWI instruction, and therefore sets the ERET auxiliary register to the address of the SWI instruction itself.
- The ARCV2 architecture defines an SWI_S instruction to allow software breakpoints to be inserted in place of the 16-bit instructions.
- The ARC EM processors with ARCOVER ≥ 0x44 always include the EFA register. The ARC EM family of processors with ARCOVER < 0x44 include the EFA register only when any of the following protections are configured: MMU, MPU, code protection, stack checking, ECC, or parity protection. The EFA register is always included in the ARC HS family of processors and processors based on the V1-700 architecture whereas the V1-600 architecture does not define this register.

- The ARCV2 architecture does not include the BTA_L1 or BTA_L2 aux registers. The V1-700 architecture includes these registers. The V1-600 architecture does not define these registers.

[Table A-1](#) and [Table A-2](#) describe the known differences in the baseline and optional architecture features between ARCV2, ARCompact for ARC 600, and ARCompact for ARC 700.

Table A-1 Baseline Differences: ARC 600, ARC 700, and ARCV2

Baseline ISA Features	ARC 600	ARC 700	ARCV2
RTIE	No	Yes	Yes
J.F [ilink] returns from interrupt	Yes	Yes	No (use RTIE)
EX	No	Yes	Yes
TRAP0	No	Yes	No
TRAP_S	No	Yes	Yes
SWI	Yes	No	Yes
AEX	No	No	Yes
SETI	No	No	Yes
CLRI	No	No	Yes
SWI_S	No	No	Yes
BRK	No	Yes	Yes
DIVAW	Yes	Yes	No
Saturating ALU and shift operations, Condition codes 0x10, 0x11	No	Yes	No
UNIMP_S	No	Yes	Yes
User / Kernel modes	No	Yes	Yes
Full exception handling Privileged instructions Protected state	No	Yes	Yes
BTA	No	Yes	Yes
BTA_L1	No	Optional	No
BTA_L2	No	Optional	No
EFA auxiliary register	No	Yes	Baseline
STATUS register (obsolete)	Yes Not in 601	Yes	No

Table A-1 Baseline Differences: ARC 600, ARC 700, and ARCV2 (Continued)

Baseline ISA Features	ARC 600	ARC 700	ARCV2
Semaphore register	Yes Not in 601	No	No
16 Register Option	Yes	No	Yes
Exception on access to unimplemented core register	No	Yes	Yes
Exception on LD, EX, POP, or other multi-cycle operation with LP_COUNT as destination	No	Yes Privilege Violation	Yes Instruction Error exception
Protected access to LP_COUNT	No	Yes	Yes
Exception on attempting to write to PCL	No (undefined behavior)	Yes	Yes
PCL usable as source registers in BRcc/BBITn	No	Yes	Yes
Reading a non-existent auxiliary register	Returns the IDENTITY register value	Instruction Error exception	Instruction Error exception
Writing a non-existent auxiliary register	Write is ignored	Instruction Error exception	Instruction Error exception
Reading a write-only aux register	Returns IDENTITY register value	Instruction Error exception	Instruction Error exception
Writing a read-only aux register	Write is ignored	Instruction Error exception	Instruction Error exception
Reading a non-existent BCR register	Returns 0	Return 0 in kernel mode, Privilege Violation Exception in user mode	Return 0 in kernel mode, Privilege Violation Exception in user mode
Load with L IMM destination (prefetch)	No	Yes	Yes
Load/Store data size mode ZZ = 11	Undefined behavior	Instruction Error exception	Instruction Error exception
Load-extend mode X = 1, when size ZZ = 00 (32-bit word)	Ignored	Instruction Error exception	Instruction Error exception
Illegal combinations of fields	Ignored	Instruction Error exception	Instruction Error exception
Use of undefined condition codes	Always returns FALSE	Instruction Error exception	Instruction Error exception

Table A-1 Baseline Differences: ARC 600, ARC 700, and ARCV2 (Continued)

Baseline ISA Features	ARC 600	ARC 700	ARCV2
BRcc sub-opcodes 0x6 to 0xD	Not defined by the PRM	Instruction Error exception	Used to encode static branch predictions
Branch in delay slot, LPcc in delay slot, RTIE in delay slot, LIMM in delay slot instruction	Undefined behavior!	Illegal Instruction Sequence exception	Illegal Instruction Sequence exception
FLAG is a serializing instruction	No	Yes	Yes
KFLAG instruction supported	No	No	Yes
SLEEP has placement restrictions	Yes	No	No
LR or SR with operand mode 0x3	Undefined behavior	Instruction Error Exception	Instruction Error Exception
w6 operand format for storing an immediate 6-bit value	No	No	Yes
Misaligned data memory exceptions	Detected, but not prevented	Protection Violation exception	Misaligned Access exception (new)
Misaligned data memory accesses	Not supported	Not supported	Supported

Table A-2 ISA Option Differences: ARC 600, ARC 700, and ARCV2

Optional ISA Features	ARC 600	ARC 700	ARCV2
EX	No	Baseline	Baseline
EFA auxiliary register	No	Baseline	Baseline
MUL64 MULU64 MUL64_S MLO, MMID, MHI, and MULHI extension core registers	Optional	No	No
MPY MPYU MPYH MPYHU	No	Optional	Optional
MPY_S	No	No	Optional
MPYW MPYUW	Optional	No	Optional

Table A-2 ISA Option Differences: ARC 600, ARC 700, and ARCV2 (Continued)

Optional ISA Features	ARC 600	ARC 700	ARCV2
DIV DIVU REM REMU	No	No	Optional
Divide-by-zero exception and STATUS32.DZ	No	No	Optional
NORM NORMW	Optional	Optional	Optional
SWAP	Optional	Optional	Optional
SWAPE	No	Baseline	Optional
LDI, LDI_S	No	No	Optional
LSL16, LSR16	No	No	Optional
Barrel shifter	Optional	Optional	Optional
Shift-assist instructions ASR16, ASR8, LSR8, LSL8, ROL8, ROR8	No	No	Optional
SETcc instructions: SETEQ, SETNE, SETLT, SETGE, SETLO, SETHS, SETLE, SETGT	No	No	Optional
ENTER_S, LEAVE_S	No	No	Optional
EI_S	No	No	Optional
JLI_S	No	No	Optional
BI, BIH	No	No	Optional
FFS, FLS	No	No	Optional

Table A-3 ISA Features: ARC EM and ARC HS

ISA Feature	ARC EM	ARC HS3x	ARC HS4x
Multiply and accumulate extension instructions: QMPYH, QMPYHU, DMPYWH, DMPYWHU, QMACH, QMACHU, DMACWH, DMACWHU,	No	Optional Select these instructions using the mpy_option = {7,8,9} option.	Optional Select these instructions using the mpy_option = {7,8,9} option.
Vector addition and subtraction instructions: VADD4H, VSUB4H, VADDSUB4H, VSUBADD4H, VADD2, VSUB2, VADDSUB, VSUBADD	No	Optional Select these instructions using the mpy_option = {7,8,9} option.	Optional Select these instructions using the mpy_option = {7,8,9} option.
DSP Extension Instructions	Optional Select these instructions using the has_dsp==1 or dsp_complex==1	No	Optional Select these instructions using the has_dsp==true
DSP Extension Instructions for Complex Numbers	Optional Select these instructions using the dsp_complex==1	No	Optional Select these instructions using the has_dsp==true
DSP wide extension instructions	No	No	Optional Select these instructions using the dsp_wide==true
LPcc: branch or jump in last position within loop body	Affects loop count	Disallowed: triggers illegal instruction sequence exception (see LPcc instruction)	Disallowed: triggers illegal instruction sequence exception (see LPcc instruction)
Double-word 64-bit operations	No	Optional, Enable/disable 64-bit operations using ll64_option = {false,true}. LDD and STD are load and store operations for 64-bit data. When atomic operation are also enabled, LLOCKD and SCONDD instructions are enabled.	Optional, Enable/disable 64-bit operations using ll64_option = {false,true}. LDD and STD are load and store operations for 64-bit data. When atomic operation are also enabled, LLOCKD and SCONDD instructions are enabled.

Table A-3 **ISA Features: ARC EM and ARC HS**

ISA Feature	ARC EM	ARC HS3x	ARC HS4x
r58 and r59	Extension core registers When <code>has_dsp==true</code> , r58 is used as an Accumulator Low register in little-endian systems and Accumulator High in big-endian systems, and r59 is used as an Accumulator High register in little-endian systems and Accumulator Low register in big-endian systems. When a floating-point unit is included and the FSMADD and FSMSUB instructions are enabled, the two extension core registers (r58 and r59) are included and comprise the third operand.	When <code>mpy_option > 6</code> , r58 is used as an Accumulator Low register in little-endian systems and Accumulator High in big-endian systems, and r59 is used as an Accumulator High register in little-endian systems and Accumulator Low register in big-endian systems. When <code>mpy_option > 6</code> or <code>fpu_fma_option</code> is defined, two extension core registers (r58 and r59) are included and comprise the 64-bit Accumulator (ACCH, ACCL).	When <code>mpy_option > 6</code> , r58 is used as an Accumulator Low register in little-endian systems and r59 is used as an Accumulator High register. When <code>mpy_option > 6</code> , <code>has_dsp==1</code> , or <code>fpu_fma_option</code> is defined, two extension core registers (r58 and r59) are included and comprise the 64-bit Accumulator (ACCH, ACCL).
IFQUEUE_BUILD auxiliary register	Included in this ISA and configurable.	Not included in this ISA.	Not included in this ISA.
<code>code_density_option</code>	Optional	Yes	Yes
<code>bit_scan_option</code>	Optional	Yes	Yes
<code>swap_option</code>	Optional	Yes	Yes
<code>byte_order</code>	little and big endian	little and big endian	little endian only

Table A-3 ISA Features: ARC EM and ARC HS

ISA Feature	ARC EM	ARC HS3x	ARC HS4x
ATOMIC_OPTION	No	Optional; When <code>atomic_option==true</code> , the following instructions are supported: LLOCK, SCOND, WLFC Further when <code>ll64_option==1</code> , the LLOCKD and SCONDD instructions are supported.	Optional; When <code>atomic_option==true</code> , the following instructions are supported: LLOCK, SCOND, WLFC Further when <code>ll64_option==true</code> , the LLOCKD and SCONDD instructions are supported.
Floating-point instructions	Yes	Yes	Yes
IC_FEATURE_LEVEL	Optional; allowed values are 0, 1, 2	Fixed at 2.	Fixed at 2.
DC_FEATURE_LEVEL	Optional; allowed values are 0, 1, 2	Fixed at 2.	Fixed at 2.
Degree of associativity for data caches (DC_WAYS)	Optional; allowed values are 0, 1, 2	Fixed at 2.	Fixed at 2.
Support for Memory management Unit	No	Yes	Yes
Support for L2 cache	No	Yes	Yes
Baseline instructions: PREFETCHW, DSYNC, DMB	No	Yes	Yes
Support for Shared Coherency Unit	No	Yes	Yes
Support for the Vector Unit	No	No	No
Multi-core cluster component	No	Yes	Yes
Bitstream instructions	Yes	No	No
XY instructions: MODIF, MODAPP	Yes	No	No
Address Generation Unit	Yes	No	Yes
FastMath extensions	No	Yes	Yes
Support for shared memory configuration between CSM and L2 cache	No	Yes	Yes
Conversion instructions: FS2H and FH2S	No	No	Yes

B

Implementation-dependent Behavior

B.1 EM Exception Handling

B.1.1 ECR User-mode Setting in the ARC EM Family of Cores

The U field, located in bit 30 of the ECR auxiliary register, indicates that although the processor was in user mode when an exception occurred, kernel privileges were in force at the time. This can happen when an interrupt prolog or epilog pushes or pops registers to or from the kernel stack when a user-mode process is interrupted. Kernel privileges are applied to all interrupt prolog memory operations that access the kernel stack. Upon completion of register push operations during an interrupt prolog, the processor transitions into kernel mode, but the processor technically remains in its pre-interrupt Kernel/User mode until the completion of the prolog.

B.2 Cause Codes for Memory Accesses and Data/Address Errors

The core-specific Parameter field may be defined by each core product to encode any information that is appropriate for that core, such as the type of memory accessed and whether it was an address or data error. ARC HS/EV/VPX5 processors fill these bits with zeros, whereas ARC EM inserts suitable values to encode the different types of CCM and data/address errors.

Cause of EV_MachineCheck Exception	Vector	Cause Code	Parameter
Internal Memory Error on Instruction Fetch	0x03	0x04	0x00
Internal Memory Error on Data Access	0x03	0x05	0x00
Illegal Overlapping MPU Entries	0x03	0x06	0x00
Secure vector table in normal memory	0x03	0x10	0x00
NSC jump table not in Secure memory	0x03	0x11	0x00

B.3 Exception Handling

B.3.1 ECR User-mode Setting for the ARC HS/EV/VPX5 Family of Cores

The ECR U bit indicates that the user/kernel mode in force at the time of the exception was different from the value of the STATUS32.U bit. This can happen when an exception is raised within an interrupt prolog or epilog where the operating mode may have been changed speculatively so that kernel privileges can apply to memory accesses to the kernel stack, or so that user privileges can apply to memory accesses to the user stack. If an exception occurs when a speculative user/kernel mode is in force, the value captured by ERSTATUS.U still reflects the committed value of STATUS32.U, and is not influenced by the speculative kernel/user mode state. An exception handler can determine the actual kernel/user mode in force at the time of taking an exception by taking the negation of ERSTATUS.U if ECR.U is set to 1.

B.4 Result on Overflow from Integer Division

The DIV, DIVU, REM and REMU instructions specify that their result is implementation dependent when overflow occurs and overflow exceptions are disabled. When overflow occurs, due to division by zero, or when dividing -2^{31} by -1 , there is no meaningful result that can be returned. Therefore, the result of such a division operator should not be relied upon by software.

Table B-1 Implementation Behavior on Division Overflow

Core	Conditions	Core-specific behavior
ARC EM	Overflow and STATUS32.DZ == 0	No update to the destination register
ARC HS/EV/ARC VPX5		Write 0 to the destination register

C

ARConnect Commands Details

This chapter describes the details of each ARConnect command and the related `CONNECT_WDATA` and `CONNECT_READBACK` registers formats. If an ARConnect command does not use the `CONNECT_WDATA` and `CONNECT_READBACK` registers or does not require any parameters, the respective sections for that command are marked as N/A (not applicable).



Note

The commands sent to ARConnect are ignored under the following conditions, and no illegal command or error information is generated from the hardware.

- If a ARConnect component (such as the inter-core interrupt unit) is not configured during build time, the commands related to this unit are ignored.
- If undefined commands are sent to an ARConnect component, the undefined commands are ignored.

For commands that do not require a parameter, the best practice is to set the parameter field to 0; for commands that require a parameter, the best practice is to set the value of the parameter in the valid range. In both cases, if the parameter is set beyond the scope or with an invalid value, the hardware result is unpredictable.

Software must guarantee the correct usage of commands for ARConnect under different configurations.

C.1 CMD_CHECK_CORE_ID

Field	Value
CONNECT_CMD[7:0]	0x00

A processor core connected to ARConnect is numbered with a `CORE_ID` ranging from 0 to $N-1$ depending on the number of cores connected to ARConnect. Each core can use the `CMD_CHECK_CORE_ID` command to find out its `CORE_ID`. The `CONNECT_READBACK` register returns the `CORE_ID`.



Note

If the external master issues the `CMD_CHECK_CORE_ID` command to ARConnect through ASI, the command is treated as a NOP.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure C-1 CONNECT_READBACK Register Format for CMD_CHECK_CORE_ID Command

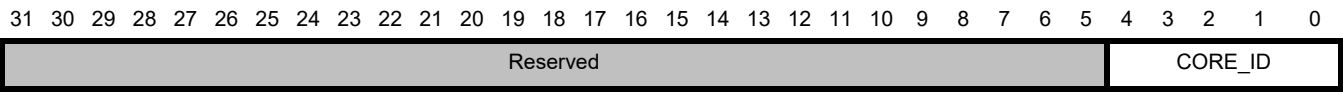


Table C-1 CONNECT_READBACK Register Field Descriptions for CMD_CHECK_CORE_ID

Field	Bits	Description
CORE_ID	[4:0]	Indicates core ID of the current core issuing CMD_CHECK_CORE_ID command 0 = CORE0 1 = CORE1 3 = CORE3

C.2 CMD_INTRPT_GENERATE_IRQ

Field	Value
CONNECT_CMD[7:0]	0x01

Each core can be an interrupt initiator core and use the CMD_INTRPT_GENERATE_IRQ command to generate an interrupt to another core or to external system in the system.

 **Note**

If the external master issues the CMD_INTRPT_GENERATE_IRQ command to ICI through ASI, the command is treated as a NOP.

The PARAMETER field specified with this command is used to distinguish whether the interrupt is generated to a core or to the external system, and this field specifies the CORE_ID of the interrupt receiver core if the interrupt is generated to the core.

Each core is assigned an internal INTRPT_STATUS register in ICI to record the status of the interrupt (acknowledged or not) from it to all other cores. The CMD_INTRPT_GENERATE_IRQ command sets one bit in the interrupt initiator core's INTRPT_STATUS register corresponding to the interrupt receiver core.

The interrupt line of the interrupt receiver core is asserted after ARConnect issues this command and is asserted until the interrupt receiver core acknowledges all the inter-core interrupts it received.

Parameter Format

The bit [8] of the parameter field defines whether the interrupt is generated to the core or to the external system.

- 1: generate a pulse-sensitive interrupt to external system, and any value in the parameter [7:0] field is ignored.
- 0: generate a level-sensitive interrupt to another core.

The Parameter [4:0] field specifies the `CORE_ID` of the receiver core. For a four-core system, set its value to 0, 1, 2, or 3.



Note

- Indexing to the core itself is meaningless and is ignored. For example, if the CORE ID of current core is 3, setting the parameter field equals 3 (Parameter [7:0]=3) accompany with this command is meaningless and no inter-core interrupt is generated.
- In single-core configuration, the ICI can still be used to generate interrupt to external system.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.3 CMD_INTRPT_GENERATE_ACK

Field	Value
CONNECT_CMD[7:0]	0x02

Each core can be an interrupt receiver core and can use the `CMD_INTRPT_GENERATE_ACK` command to generate an interrupt acknowledgement to the interrupt-initiating core.



Note

- If the external master issues the `CMD_INTRPT_GENERATE_ACK` command to ICI through ASI, the command is treated as a NOP.

The `PARAMETER` field is used to specify the `CORE_ID` of the interrupt initiator core.

The `CMD_INTRPT_GENERATE_ACK` command clears the bit that corresponds to the interrupt receiver core in the interrupt initiator core's `INTRPT_STATUS` register. However, the interrupt line of this interrupt receiver core is only de-asserted after it serves all the inter-core interrupts it has received.

Parameter Format

The `Parameter [4:0]` field specifies the `CORE_ID` of the interrupt initiator core. For a four-core system, set its value to 0, 1, 2, or 3.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.4 CMD_INTRPT_READ_STATUS

Field	Value
CONNECT_CMD[7:0]	0x03

As an interrupt initiator, each core can use the `CMD_INTRPT_READ_STATUS` command to check the interrupt status generated from it to other cores.

 **Note**

If the external master issues the `CMD_INTRPT_READ_STATUS` command to ICI through ASI, the command is treated as a NOP.

The `PARAMETER` field is used to specify the `CORE_ID` of the interrupt receiver core. The `CONNECT_READBACK` register returns the value of the bit corresponding to the interrupt receiver core from the interrupt initiator's `INTRPT_STATUS` register.

Parameter Format

The `Parameter [4:0]` field specifies the `CORE_ID` of the interrupt receiver core. For a four-core system, set its value to 0, 1, 2, or 3.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

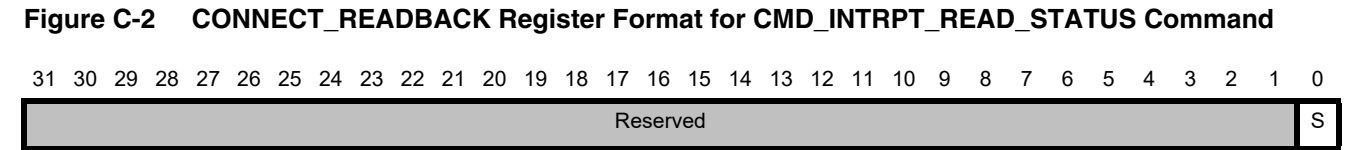


Table C-2 CONNECT_READBACK Register Field Descriptions for CMD_INTERRUPT_READ_STATUS

Field	Bits	Description
S	[0]	Interrupt Status of the specified inter-core interrupt 0 = there is no outstanding interrupt or a former interrupt has been serviced (acknowledged) 1 = there is an outstanding interrupt

C.5 CMD_INTRPT_CHECK_SOURCE

Field	Value
CONNECT_CMD[7:0]	0x04

All the inter-core interrupts from other cores to the interrupt receiver core are combined into single interrupt line. An interrupt receiver core can use the CMD_INTRPT_CHECK_SOURCE command to find the source of an inter-core interrupt.



Note

If the external master issues the CMD_INTRPT_CHECK_SOURCE command to ICI through ASI, the command is treated as a NOP.

The CONNECT_READBACK register returns a vector gathered from the internal INTRPT_STATUS registers of all interrupt initiator cores.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

For a four-core system, the CONNECT_READBACK register returns a vector value after the core issues the CMD_INTRPT_CHECK_SOURCE command.

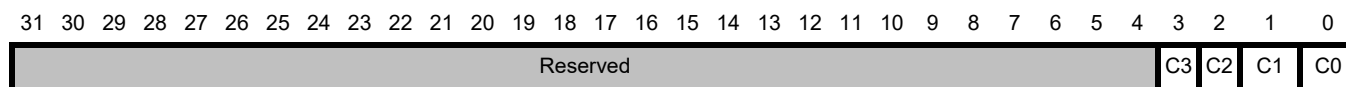
Figure C-3 CONNECT_READBACK Register Format for CMD_INTRPT_CHECK_SOURCE Command

Table C-3 CONNECT_READBACK Register Field Descriptions for CMD_INTRPT_CHECK_SOURCE

Field	Bits	Description
C0	[0]	Indicates whether Core0 is generating an interrupt to the current core. 0 = Core0 is not generating an interrupt to the current core. 1 = Core0 is generating an interrupt to the current core.
C1	[1]	Indicates whether Core1 is generating an interrupt to the current core. 0 = Core1 is generating an interrupt to the current core. 1 = Core1 is generating an interrupt to the current core.
C2	[2]	Indicates whether Core2 is generating an interrupt to the current core. 0 = Core2 is not generating an interrupt to the current core. 1 = Core2 is generating an interrupt to the current core.
C3	[3]	Indicates whether Core3 is generating interrupt to the current core. 0 = Core3 is not generating an interrupt to the current core. 1 = Core3 is generating an interrupt to the current core.

C.6 CMD_INTRPT_GENERATE_ACK_BIT_MASK

Field	Value
CONNECT_CMD[7:0]	0x05

Each core can be an interrupt receiver core and can use the CMD_INTRPT_GENERATE_ACK_BIT_MASK command to generate an interrupt acknowledgment to the interrupt-initiating core.



If an external master issues this command to ICI through ASI, the command is treated as a NOP.

The PARAMETER field is used to specify the bit mask of the interrupt initiator core.

The CMD_INTRPT_GENERATE_ACK_BIT_MASK command clears the bit that corresponds to the interrupt receiver core in the interrupt initiator core's INTRPT_STATUS register. However, the interrupt line of this interrupt receiver core is only de-asserted after it serves all the inter-core interrupts it has received.

**Note**

CMD_INTRPT_GENERATE_ACK_BIT_MASK” can set multi-core bit mask to clear 1 or more initiator core’s INTRPT_STATUS register while “CMD_INTRPT_GENERATE_ACK” can only set 1 core.

Interrupt handler can use “CMD_INTRPT_CHECK_SOURCE” to get bit mask information for which cores are set the interrupt to the destination core, if the interrupt handler wants to clear all the initiator cores interrupt status simultaneously, it can use the bit mask result getting from CMD_INTRPT_CHECK_SOURCE to set CMD_INTRPT_GENERATE_ACK_BIT_MASK and clear all the core interrupt at once.

Parameter Format

The parameter [15:0] field specifies the core bit mask of ARC core of the interrupt initiator core, so this commands can be supported in 16 core systems at most.

Each bit[n] is used to indicates each interrupt initiator core whether should be cleared the interrupt INTRPT_STATUS register or not.

- 1: corresponding interrupt initiator core INTRPT_STATUS register is cleared.
- 0: corresponding interrupt initiator core INTRPT_STATUS register is NOT cleared

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.7 CMD_INTRPT_EXT_MODE

Field	Value
CONNECT_CMD[7:0]	0x06

Each core or the external master can use the CMD_INTRPT_EXT_MODE command to program the hardware behavior of interrupt signal to external host.

Parameter Format

The parameter [15:0] field specifies the interrupt hardware behavior mode:

- 1: Generate level-sensitive interrupt to host.
- 0: Generate pulse-sensitive interrupt to host.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.8 CMD_INTRPT_SET_PULSE_CNT

Field	Value
CONNECT_CMD[7:0]	0x07

An internal counter PULSE_CNT is implemented in the ICI component to control the number of cycles asserted for a pulse-sensitive interrupt to external host. Each core or external master can use the CMD_INTRPT_SET_PULST_CNT command to set the PULSE_CNT.

 **Note**

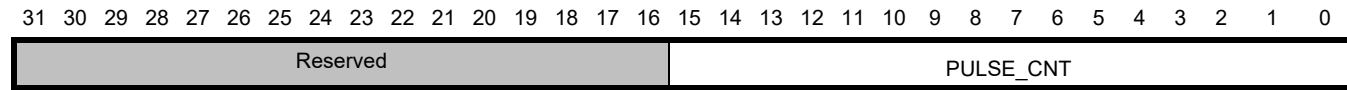
The cycle stands from MCIP clock domain perspective. That is, if the PULSE_CNT is set as 5, then the pulse sensitive interrupt lasts for 5 MCIP clock cycles.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure C-4 CONNECT_WDATA Register Format for CMD_INTRPT_SET_PULSE_CNT Command



 **Note**

The least high duration is 2 MCIP clock cycles.

CONNECT_READBACK Register Format

N/A

C.9 CMD_INTRPT_READ_PULSE_CNT

Field	Value
CONNECT_CMD[7:0]	0x08

Each core or the external master can use the CMD_INTRPT_READ_PULSE_CNT command to read the value of pulse duration counter in the ICI unit.

The CONNECT_READBACK register returns the content of pulse interrupt duration register.

Parameter Format

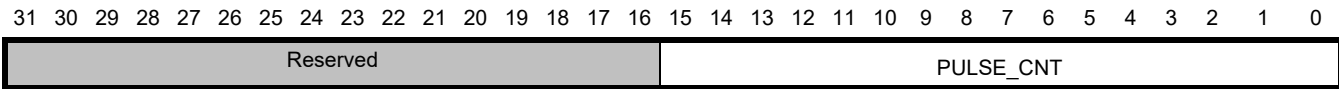
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure C-5 CONNECT_READBACK Register Format for CMD_INTRPT_READ_PULSE_CNT Command



C.10 CMD_SEMA_CLAIM_AND_READ

Field	Value
CONNECT_CMD[7:0]	0x11

Each core or the external master can use the CMD_SEMA_CLAIM_AND_READ command to claim the specified semaphore and verify its ownership. Specify the index of the semaphore in the PARAMETER field.

The CONNECT_READBACK register returns whether the claim is a success or not.

Parameter Format

The Parameter [4:0] field defines the index of the semaphore; Set its value from 0, 1, to N-1 where N is the number of semaphores configured at the system build-time.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure C-6 CONNECT_READBACK Register Format for CMD_SEMA_CLAIM_AND_READ Command

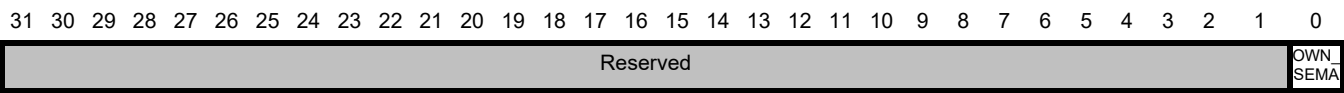


Table C-4 CONNECT_READBACK Register Field Descriptions for CMD_SEMA_CLAIM_AND_READ

Field	Bits	Description
OWN_SEMA	[0]	Semaphore ownership status 0 = the semaphore is not available to the core, and the claim request has failed. 1 = the core has successfully obtained the semaphore.

C.11 CMD_SEMA_RELEASE

Field	Value
CONNECT_CMD[7:0]	0x12

If the core or the external master has already obtained the ownership of a semaphore, to release the semaphore and allow it to be available for other cores, the core can use the `CMD_SEMA_RELEASE` command.

The `PARAMETER` field specifies the index of the semaphore.

Parameter Format

The `Parameter [4:0]` field defines the index of the semaphore; Set its value from 0, 1, to N-1 where N is the number of semaphores configured in the `CONNECT_SEMA_BUILD` register during the system build-time.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.12 CMD_SEMA_FORCE_RELEASE

Field	Value
CONNECT_CMD[7:0]	0x13

Each core or the external master can use the `CMD_SEMA_FORCE_RELEASE` command to compulsively release and make available a semaphore that has been occupied by other cores in the system.

The `PARAMETER` field specifies the index of the semaphore.

Parameter Format

The `Parameter [4:0]` field defines the index of the semaphore; Set its value from 0, 1, to N-1 where N is the number of semaphores configured in the `CONNECT_SEMA_BUILD` register during the system build-time.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.13 CMD_MSG_SRAM_SET_ADDR

Field	Value
CONNECT_CMD[7:0]	0x21

The message-passing SRAM is an ARConnect local memory. Each core can use the `CMD_MSG_SRAM_SET_ADDR` command to set the address of message-passing SRAM for subsequent read or write SRAM operations.

Each core is assigned an internal `MSG_ADDR` register in ICM to record the address of message-passing SRAM for it. The `CONNECT_WDATA` register contains the data to be written into the internal `MSG_ADDR` register.

Parameter Format

N/A

CONNECT_WDATA Register Format**Figure C-7 CONNECT_WDATA Register Format for CMD_MSG_SRAM_SET_ADDR Command**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																SRAM_ADDR[11:0]											Reserved				

**Note**

Read or write access to the message-passing SRAM is word-aligned, so the two least-significant bits of `SRAM_ADDR` are always ignored. And, the address exceeding the size of the SRAM field is truncated automatically.

C.14 CMD_MSG_SRAM_READ_ADDR

Field	Value
CONNECT_CMD[7:0]	0x22

Each core or the external master can use the `CMD_MSG_SRAM_READ_ADDR` command to read the value of its internal `MSG_ADDR` register.

The `CONNECT_READBACK` register returns the value of the internal `MSG_ADDR` register of the core.

Parameter Format

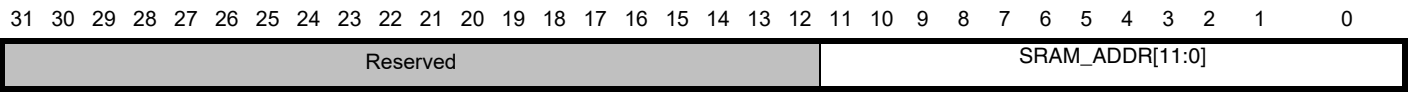
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure C-8 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_ADDR Command



C.15 CMD_MSG_SRAM_SET_ADDR_OFFSET

Field	Value
CONNECT_CMD[7:0]	0x23

Each core or the external master can use the CMD_MSG_SRAM_SET_ADDR_OFFSET command to set the address offset value of the message-passing SRAM for subsequent read or write SRAM operations in incremental mode.

Each core and the external master is assigned an internal MSG_ADDR_OFFSET register in ICM to record the address offset value for the core.

The value of the CONNECT_WDATA register is used with the CMD_MSG_SRAM_SET_ADDR_OFFSET command, to update the internal MSG_ADDR_OFFSET register.

Parameter Format

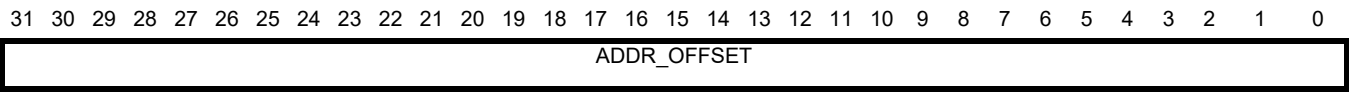
N/A

CONNECT_WDATA Register Format

The offset data must be written into the CONNECT_WDATA register before the CMD_MSG_SRAM_SET_ADDR_OFFSET command is issued as shown in [Figure C-9](#).

The offset value is a signed value and encoded with 2’s complement, which ranges from -64 to +60. This value is passed to ICM for subsequent incremental read or write SRAM operations.

Figure C-9 CONNECT_WDATA Register Format for Set-Offset Command



 **Note**

- The offset value must be word aligned, so the two least-significant bits of ADDR_OFFSET are always ignored.
- The hardware result is unpredictable if the written offset value exceeds the allowed range.

CONNECT_READBACK Register Format

N/A

C.16 CMD_MSG_SRAM_READ_ADDR_OFFSET

Field	Value
CONNECT_CMD[7:0]	0x24

Each core or the external master can use the CMD_MSG_READ_SRAM_ADDR_OFFSET command to read its address offset value.

The CONNECT_READBACK register returns the value of the internal MSG_ADDR_OFFSET register of the core.

Parameter Format

N/A

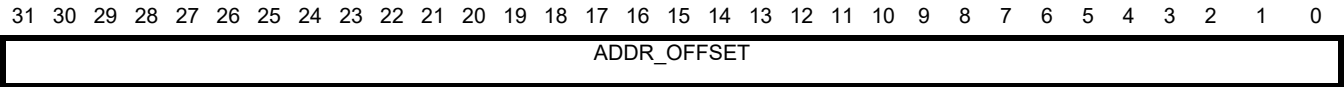
CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the SRAM-address offset value from ICM set by the last CMD_MSG_SRAM_SET_ADDR_OFFSET command. The offset value is a signed value and encoded with 2’s complement, which ranges from -64 to +60, as shown in [Figure C-10](#).

Figure C-10 CONNECT_READBACK Register Format for Read-Offset Command



C.17 CMD_MSG_SRAM_WRITE

Field	Value
CONNECT_CMD[7:0]	0x25

Each core or the external master can use the CMD_MSG_SRAM_WRITE command to trigger a write operation to the message-passing SRAM.

The address of message-passing SRAM is determined by the internal MSG_ADDR register (which can be updated using the CMD_MSG_SRAM_SET_ADDR command, the CMD_MSG_SRAM_WRITE_INC command, or the CMD_MSG_SRAM_READ_INC command).

The CONNECT_WDATA register contains the data to be written into this specified address.

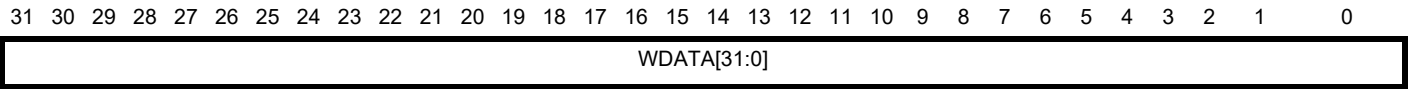
Parameter Format

N/A

CONNECT_WDATA Register Format

The CONNECT_WDATA register contains the data to be written into the specified address of SRAM before the write-data commands are sent to the message-passing SRAM.

Figure C-11 CONNECT_WDATA Register Format for CMD_MSG_SRAM_WRITE Command



CONNECT_READBACK Register Format

N/A

C.18 CMD_MSG_SRAM_WRITE_INC

Field	Value
CONNECT_CMD[7:0]	0x26

To facilitate streamlined write operations to consecutive or regular addresses in the message-passing SRAM, an incremental write mode is supported in ICM.

After setting the address offset value and the start access address, each core can use the CMD_MSG_SRAM_WRITE_INC command to continuously write data into the message-passing SRAM without specifying the access address each time.

This command implicitly updates the internal MSG_ADDR register by (current MSG_ADDR value + MSG_ADDR_OFFSET value) after execution.

The CONNECT_WDATA register contains the data to be written into current address of message-passing SRAM.

Parameter Format

N/A

CONNECT_WDATA Register Format

The CONNECT_WDATA register must contain the data to be written into the specified address of SRAM before write-data commands are sent to the message passing SRAM.

The layout of the `CONNECT_WDATA` is shown in [Figure C-11](#).

CONNECT_READBACK Register Format

N/A

C.19 CMD_MSG_SRAM_WRITE_IMM

Field	Value
CONNECT_CMD[7:0]	0x27

Each core or the external master can use the `CMD_MSG_SRAM_WRITE_IMM` command to write data into the message passing SRAM with the address specified by an immediate value.

The Parameter field specifies this immediate address value. The value must be word aligned, so the two least-significant bits are always ignored.



The `CMD_MSG_SRAM_WRITE_IMM` command does not update the value of core's `MSG_ADDR` register.

Parameter Format

The `Parameter[11:0]` field specifies the SRAM address.

CONNECT_WDATA Register Format

The `CONNECT_WDATA` register must contain the data to be written into the specified address of SRAM before write-data commands are sent to the message passing SRAM.

The layout of the `CONNECT_WDATA` register is shown in [Figure C-11](#).

CONNECT_READBACK Register Format

N/A

C.20 CMD_MSG_SRAM_READ

Field	Value
CONNECT_CMD[7:0]	0x28

Each core or the external master can use `CMD_MSG_SRAM_READ` command to trigger a read operation to the message passing SRAM.

The address of message passing SRAM is determined by the internal `MSG_ADDR` register (which can be updated by the `CMD_MSG_SRAM_SET_ADDR` command, the `CMD_MSG_SRAM_WRITE_INC` command, or the `CMD_MSG_SRAM_READ_INC` command).

The `CONNECT_READBACK` register contains the data read out from this specified address.

Parameter Format

N/A

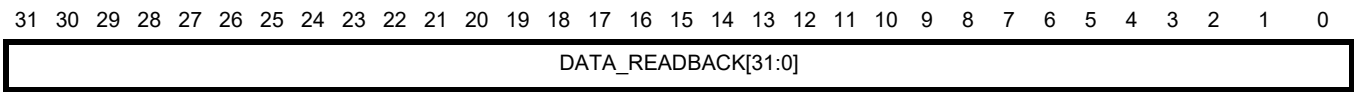
CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the data read from specified address of SRAM for the `CMD_MSG_SRAM_READ` command.

Figure C-12 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ Command



C.21 CMD_MSG_SRAM_READ_INC

Field	Value
CONNECT_CMD[7:0]	0x29

To facilitate streamlined read operations to consecutive or regular addresses in the message-passing SRAM, an incremental read mode is supported in ICM.

After setting the address offset value and the start access address, each core or the external master can use the `CMD_MSG_SRAM_READ_INC` command to continuously read the data from the message-passing SRAM without specifying the access address each time.

This command implicitly updates the internal `MSG_ADDR` register (current `MSG_ADDR` value + `MSG_ADDR_OFFSET` value) after execution.

The `CONNECT_READBACK` register contains the data read out from current address of the message-passing SRAM.

Parameter Format

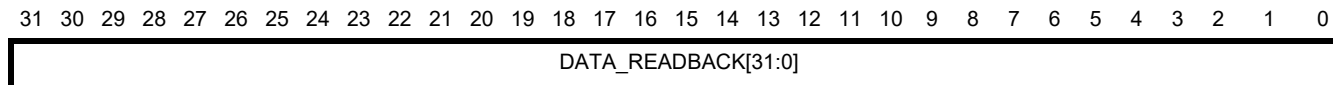
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the data read from the specified address of SRAM for the `CMD_MSG_SRAM_READ_INC` command.

Figure C-13 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_INC Command

C.22 CMD_MSG_SRAM_READ_IMM

Field	Value
CONNECT_CMD[7:0]	0x2A

Each core or the external master can use the CMD_MSG_SRAM_READ_IMM command to read the data from the message-passing SRAM using the immediate address. The PARAMTER field specifies the immediate address.



Note

The CMD_MSG_SRAM_READ_IMM command does not update the core's MSG_ADDR register.

Parameter Format

The Parameter [11:0] field specifies the SRAM address by an immediate.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the data read from the specified address of SRAM. The layout of CONNECT_READBACK register is same as of the CMD_MSG_SRAM_READ command.

C.23 CMD_MSG_SET_ECC_CTRL

Field	Value
CONNECT_CMD[7:0]	0x2B

ICM has an internal MSG_ECC_CTRL register to enable or disable ECC or parity protection on the Message SRAM. Additionally, flag bits (16 or 17) are updated in the MSG_ECC_CTRL register when protection is enabled and non-correctable error occurs.

Each core or external master can use the CMD_MSG_SET_ECC_CTRL command to set the MSG_ECC_CTRL register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The data must be written to the `CONNECT_WDATA` register before issuing the `CMD_MSG_SET_ECC_CTRL` command.

Figure C-14 `CONNECT_WDATA` Register Format for `CMD_MSG_SET_ECC_CTRL` Command

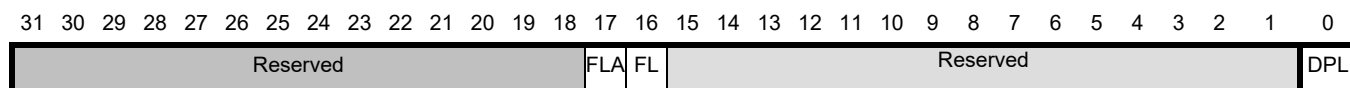


Table C-5 `CONNECT_WDATA` Register Field Description for `CMD_MSG_SET_ECC_CTRL`

Field	Bits	Description
DPL	[0]	Control bit for ECC/parity protection on the Message SRAM; 0 = enable protection 1 = disable protection On reset, the control bits are set to 0; that is ECC/parity protection is enabled. NOTE: If protection is disabled, the FL flag is never set. When ECC or parity logic is disabled and will never detect and correct errors. NOTE: If ECC is not configured, the value of the DPL bit is always 1, writes are ignored.
FL	[16]	Error Flag 0 : No ECC error detected 1 : Single-bit error (in Parity configuration) or 2-bits errors in data or address (in SECCDED configuration) and 2-bit error in address NOTE: The FL bit can only be set by the ECC hardware logic; it cannot be set by software. But software can use the <code>CMD_MSG_SET_ECC_CTRL</code> command to write 0 to clear this bit. And software can also use the <code>CMD_MSG_READ_ECC_CTRL</code> to read the FL bit. If the software writes 0 and ECC hardware logic also sets this bit simultaneously, the software will preempt.

Table C-5 CONNECT_WDATA Register Field Description for CMD_MSG_SET_ECC_CTRL

Field	Bits	Description
FLA	[17]	Address single-bit error flag. The FLA bit can only be asserted by the ECC/Parity hardware logic; it cannot be asserted by software. software can use the CMD_MSG_SET_ECC_CTRL command to write 0 to clear this bit, and use the CMD_MSG_READ_ECC_CTRL command to read the FLA value. 0: No error detected 1: Single-bit ECC or parity error in address To clear this flag, the software must write 0 to this bit. If the software writes 0 and ECC hardware logic also sets this bit simultaneously, the software will preempt.

CONNECT_READBACK Register Format

N/A.

C.24 CMD_MSG_READ_ECC_CTRL

Field	Value
CONNECT_CMD[7:0]	0x2C

Each core or external master can use the CMD_MSG_READ_ECC_CTRL command to read the value of the internal MSG_ECC_CTRL register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

This command returns the value of the MSG_ECC_CTRL register. The layout of the CONNECT_READBACK register is same as the CONNECT_WDATA register for the CMD_MSG_SET_ECC_CTRL command.

C.25 CMD_MSG_READ_ECC_SBE_CNT

Field	Value
CONNECT_CMD[7:0]	0x2D

Each core or external master uses the `CMD_MSG_READ_ECC_SBE_CNT` command to read the value of the internal `ICM_SBE_CNT` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` returns the value of the internal `ICM_SBE_CNT` register as shown in [Figure C-15](#).

Figure C-15 CONNECT_READBACK Register Format for CMD_MSG_READ_ECC_SBE_CNT Command

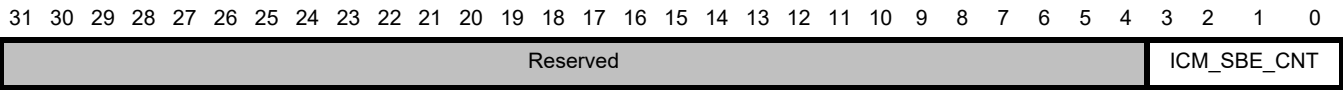


Table C-6 CONNECT_READBACK Register Field Description for CMD_MSG_READ_ECC_SBE_CNT

Field	Bits	Description
SBE_CNT	[3:0]	ECC error counter. It records the numbers of the 1-bit ECC error correction.

C.26 CMD_MSG_SRAM_CLEAR_ECC_SBE_CNT

Field	Value
CONNECT_CMD[7:0]	0x2E

Each core or external master uses the `CMD_MSG_CLEAR_ECC_SBE_CNT` command to clear the value of the internal `ICM_SBE_CNT` register to 0.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.27 CMD_DEBUG_RESET

Field	Value
CONNECT_CMD[7:0]	0x31

Each core or the external master can use the `CMD_DEBUG_RESET` command to generate reset request to cores in the system. An internal `MCD_CORE` register in ICD is used to determine the cores in the system that should be reset.

The `CONNECT_WDATA` register contains the data to be written to the internal `MCD_CORE` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The `CONNECT_WDATA` register must contain the data to be written to the internal `MCD_CORE` register before issuing the `CMD_DEBUG_RESET` command.

For a four-core system, the layout of the `CONNECT_WDATA` register is shown in [Figure C-16](#).

Figure C-16 CONNECT_WDATA Register Format for CMD_DEBUG_RESET Command

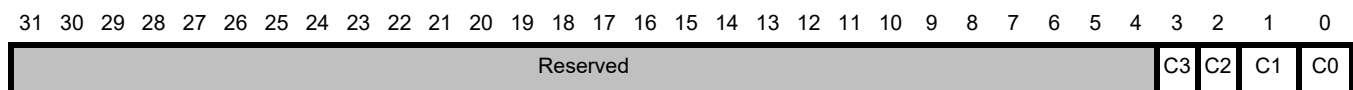


Table C-7 CONNECT_WDATA Field Descriptions for CMD_DEBUG_RESET

Field	Bits	Description
C0	[0]	Core0 select. Indicates whether core0 is selected for the current command 0 = Not selected 1 = Core0 is selected
C1	[1]	Core1select. Indicates whether core1 is selected for the current command 0 = Not selected 1 = Core1 is selected
C2	[2]	Core2 select. Indicates whether core2 is selected for the current command 0 = Not selected 1 = Core2 is selected

Table C-7 CONNECT_WDATA Field Descriptions for CMD_DEBUG_RESET

Field	Bits	Description
C3	[3]	Core3 select. Indicates whether core3 is selected for the current command 0 = Not selected 1 = Core3 is selected

CONNECT_READBACK Register Format

N/A

C.28 CMD_DEBUG_HALT

Field	Value
CONNECT_CMD[7:0]	0x32

Each core or the external master can use the `CMD_DEBUG_HALT` command to generate halt request to cores in the system. The internal `MCD_CORE` register in ICD is used to determine the cores in the system that should be halted.

The `CONNECT_WDATA` register contains the data to be written to the internal `MCD_CORE` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

CONNECT_READBACK Register Format

N/A

C.29 CMD_DEBUG_RUN

Field	Value
CONNECT_CMD[7:0]	0x33

Each core or the external master can use the `CMD_DEBUG_RUN` command to generate run request to cores in the system. The internal `MCD_CORE` register in ICD is used to determine the cores in the system that should be run.

The `CONNECT_WDATA` register contains the data to be written into the internal `MCD_CORE` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

CONNECT_READBACK Register Format

N/A

C.30 CMD_DEBUG_SET_MASK

Field	Value
CONNECT_CMD[7:0]	0x34

Each core or the external master can use the `CMD_DEBUG_SET_MASK` command to set the internal `MASK` registers in ICD. The `MASK` register determines whether a global halt is triggered if a core is halted in response to one of the following events: core halt, actionpoint halt, self-halt, and breakpoint halt.

The `PARAMETER` field is used to specify the mask bits.

Each core is assigned an internal `MASK` register. The internal `MCD_CORE` register in the ICD is used to determine the cores in the system for which the `MASK` register should be updated.

The `CONNECT_WDATA` register contains the data to be written into the internal `MCD_CORE` register.

Parameter Format

The `Parameter[3:0]` field specifies the mask bits:

- `Parameter[0]` defines whether a core halt status (whenever the `STATUS32[H]` is set) is masked or not
 - 0: Masked; no global halt triggered when core is halted
 - 1: Not masked; whenever the core is halted, a global halt is triggered
- `Parameter[1]` defines whether an actionpoint caused halt (if actionpoint option is enabled) is masked or not
 - 0: Masked; no global halt triggered when core is halted by an actionpoint
 - 1: Not masked; if an actionpoint caused halt occurs, a global halt is triggered
- `Parameter[2]` defines whether a breakpoint caused halt is masked or not
 - 0: Masked; no global halt triggered when core is halted by a breakpoint
 - 1: Not masked; if a breakpoint caused halt occurs, a global halt is triggered
- `Parameter[3]` defines whether a self-halt is masked or not
 - 0: Masked; no global halt triggered when core is halted by a self-halt
 - 1: Not masked; if a self-halt occurs, a global halt is triggered

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

CONNECT_READBACK Register Format

N/A

C.31 CMD_DEBUG_READ_MASK

Field	Value
CONNECT_CMD[7:0]	0x35

Each core or the external master can use the `CMD_DEBUG_READ_MASK` command to read its internal MASK register. The internal `MCD_CORE` register in ICD is used to determine the cores in the system that can MASK register.

The `CONNECT_WDATA` register contains the data to be written into the internal `MCD_CORE` register.

The `CONNECT_READBACK` register returns the value of internal MASK register of the specified core.

Parameter Format

N/A

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

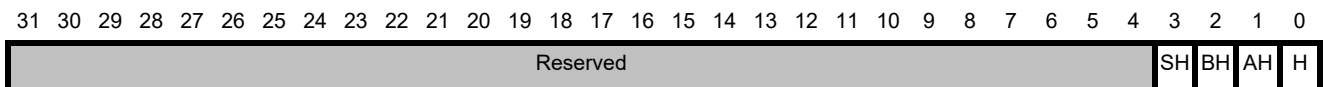


Note

The `CMD_DEBUG_READ_MASK` command supports reading of the internal MASK register of one core at a time. That is, only one bit in the `CONNECT_WDATA` can be specified with this command.

CONNECT_READBACK Register Format

Figure C-17 CONNECT_READBACK Register Format for CMD_DEBUG_READ_MASK Command



The `CONNECT_READBACK` register indicates whether each of the following events trigger a global halt: core halt (H), actionpoint halt (AH), software breakpoint halt (BH), and self halt (SH).



Note

The H bit corresponds to the H field of the `STATUS32` register of the core, and is the complement of the core's EN port.

C.32 CMD_DEBUG_SET_SELECT

Field	Value
CONNECT_CMD[7:0]	0x36

Each core or the external masters can use the `CMD_DEBUG_SET_SELECT` command to select cores that should be halted if the core issuing the command is halted.

The internal `MCD_CORE` register in ICD is used to determine the cores in the system that should be halted.

If a core wants to halt any other cores in the system in response to its own halt events, the core should issue the `CMD_DEBUG_SET_SELECT` command before issuing the `CMD_DEBUG_SET_MASK` command, so that no global halt pulse is skipped.

The `CONNECT_WDATA` register contains the data to be written into the internal `MCD_CORE` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

The layout of `CONNECT_WDATA` register is same as of `CMD_DEBUG_RESET` command.

CONNECT_READBACK Register Format

N/A

C.33 CMD_DEBUG_READ_SELECT

Field	Value
CONNECT_CMD[7:0]	0x37

Each core or the external master can use the `CMD_DEBUG_READ_SELECT` command to read the internal `SELECT` register in ICD.

The `SELECT` register is a common register for all cores, so you need not set any value in the `CONNECT_WDATA` register for core selection.

The `CONNECT_READBACK` register returns the value of internal `SELECT` register

Parameter Format

N/A

CONNECT_WDATA Register Format

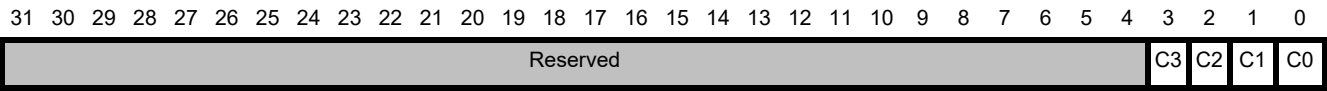
N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal SELECT register set by the previous CMD_DEBUG_SET_SELECT command.

For a four-core configuration as the example, the register layout is shown in [Figure C-18](#).

Figure C-18 CONNECT_READBACK Register Format for CMD_DEBUG_READ_SELECT Command



C.34 CMD_DEBUG_READ_EN

Field	Value
CONNECT_CMD[7:0]	0x38

Each core or the external master can use the CMD_DEBUG_READ_EN command to read the status, halt or run, of all cores in the system.

The CONNECT_READBACK register returns a vector data, each bit corresponding to the status of each core.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The CMD_DEBUG_READ_EN command returns the current state of the EN bits (the complement of the H bit in STATUS32 register) of all the cores in the system to the CONNECT_READBACK register.

For a four-core configuration, the register layout is shown in [Figure C-19](#).

Figure C-19 CONNECT_READBACK Register Format for CMD_DEBUG_READ_EN

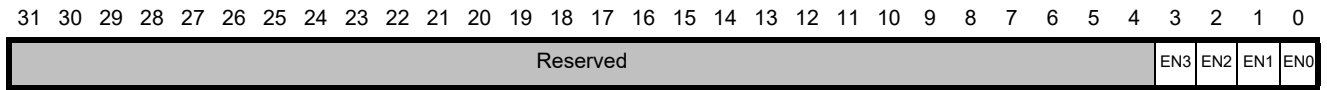


Table C-8 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_EN

Field	Bits	Description
EN[n]	[3:0]	The current status of the EN bits 1: indicates that the core is running 0: indicates that the core is halted

C.35 CMD_DEBUG_READ_CMD

Field	Value
CONNECT_CMD[7:0]	0x39

Each core or the external master can use the `CMD_DEBUG_READ_CMD` command to check the last command sent to ICD.

An internal `MCD_CMD` register in ICD records the last command and the parameter data (if needed) it received. The `CONNECT_READBACK` register reads out the value of this internal `MCD_CMD` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CMD_DEBUG_READ_CMD` command returns the last-written contents of the internal `MCD_CMD` register to the `CONNECT_READBACK` register, as shown in [Figure C-20](#).

Figure C-20 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CMD Command

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								PARAMETER[15:0] (Last written)														Command (Last written)									

C.36 CMD_DEBUG_READ_CORE

Field	Value
CONNECT_CMD[7:0]	0x3A

Each core or the debugger can use the `CMD_DEBUG_READ_CORE` command to read the value of internal `MCD_CORE` register in ICD.

The `CONNECT_READBACK` register returns the data.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CMD_DEBUG_READ_CORE` command returns the last-written contents of the `MCD_CORE` register to the `CONNECT_READBACK` register. For a four-core configuration, the layout of the `CONNECT_READBACK` register is shown in [Figure C-21](#).

Figure C-21 `CONNECT_READBACK` Register Format for `CMD_DEBUG_READ_CORE` Command

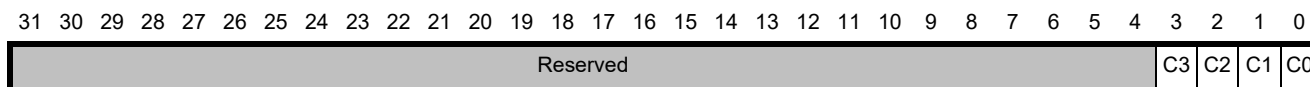


Table C-9 `CONNECT_READBACK` Register Field Descriptions for `CMD_DEBUG_READ_CORE`

Field	Bits	Description
C0	[0]	Indicates whether the core0 is selected for the debug-related command 0 = Not selected 1 = Core0 is selected
C1	[1]	Indicates whether the Core1 is selected for the debug-related command 0 = Not selected 1 = Core1 is selected
C2	[2]	Indicates whether the Core2 is selected for the debug-related command 0 = Not selected 1 = Core2 is selected
C3	[3]	Indicates whether the Core3 is selected for the debug-related command 0 = Not selected 1 = Core3 is selected

C.37 CMD_GFRC_CLEAR

Field	Value
CONNECT_CMD[7:0]	0x41

The 64-bit Global Free Running Counter (GFRC) is supported in ARConnect to enable the software to synchronize multiple independent clock domain system.

Each core or the external master can use the `CMD_GFRC_CLEAR` command to clear GFRC.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.38 CMD_GFRC_READ_LO

Field	Value
CONNECT_CMD[7:0]	0x42

Each core or the external master can use the `CMD_GFRC_READ_LO` command to read the lower 32-bits of the GFRC counter.

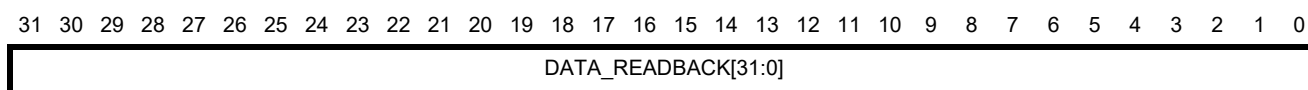
The `CONNECT_READBACK` register returns the lower 32-bit data.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format**Figure C-22 CONNECT_READBACK Register Format for CMD_GFRC_READ_LO Command****C.39 CMD_GFRC_READ_HI**

Field	Value
CONNECT_CMD[7:0]	0x43

Each core or the external master can use the `CMD_GFRC_READ_HI` command to read the higher 32-bits of the GFRC counter.

The `CONNECT_READBACK` register returns the higher 32-bit data.

Parameter Format

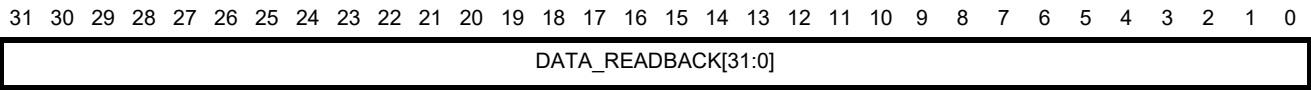
N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

Figure C-23 CONNECT_READBACK Register Format for CMD_GFRC_READ_HI Command



C.40 CMD_GFRC_ENABLE

Field	Value
CONNECT_CMD[7:0]	0x44

The GFRC have a DISABLE register to disable or enable all GFRC. Each core or external master can use the CMD_GFRC_ENABLE command to enable GFRC. The DISABLE register is zero after reset, means the GFRC is enabled by default.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.41 CMD_GFRC_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x45

The GFRC have a DISABLE register to disable or enable all GFRC. Each core or external master can use the CMD_GFRC_DISABLE command to disable GFRC. The DISABLE register is zero after reset, means the GFRC is enabled by default.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.42 CMD_GFRC_READ_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x46

Each core or external master can use CMD_GFRC_READ_DISABLE command to read the internal DISABLE register of the GFRC unit.

Parameter Format

N/A

CONNECT_WDATA Register Format

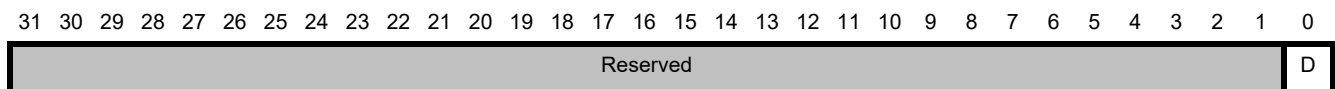
N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the content of internal DISABLE register after CMD_GFRC_READ_DISABLE command.

Figure C-24 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command.

Figure C-24 CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command

**C.43 CMD_GFRC_SET_CORE**

Field	Value
CONNECT_CMD[7:0]	0x47

The GFRC has an internal CORE register. Each core or the external master can use the CMD_GFRC_SET_CORE command to set the relevant cores to halt the GFRC. The CORE register is zero after reset.

The CONNECT_WDATA register contains the data to be written into the internal CORE register.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure C-24 shows CONNECT_WDATA Register Format for Commands Updating the internal HALT register.

Figure C-25 CONNECT_WDATA Register Format for Command CMD_GFRC_SET_CORE

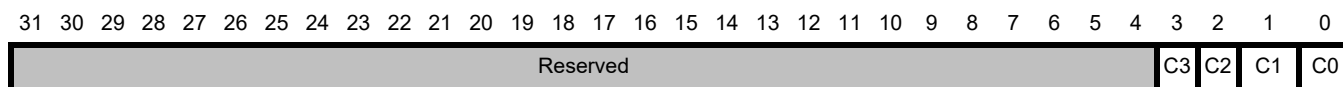


Table C-10 CONNECT_WDATA Field Descriptions

Field	Bits	Description
C0	[0]	CORE0 select Indicate CORE0 is selected for the current command 0 = Not selected 1 = CORE0 is selected
C1	[1]	CORE1 select Indicate CORE1 is selected for the current command 0 = Not selected 1 = CORE1 is selected
C2	[2]	CORE2 select Indicate CORE2 is selected for the current command 0 = Not selected 1 = CORE2 is selected
C3	[3]	CORE3 select Indicate CORE3 is selected for the current command 0 = Not selected 1 = CORE3 is selected

CONNECT_READBACK Register Format

N/A

C.44 CMD_GFRC_READ_CORE

Field	Value
CONNECT_CMD[7:0]	0x48

Each core or the external master can use the CMD_GFRC_READ_CORE command to read the internal CORE register.

The CONNECT_READBACK register returns the value of internal CORE register.

Parameter Format

N/A

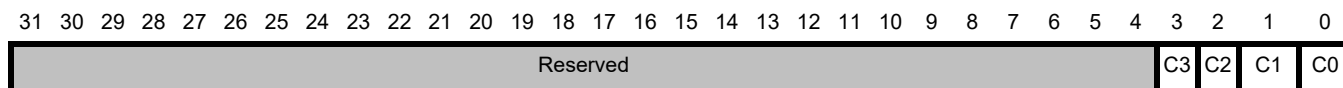
CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal CORE register.

Figure C-26 CONNECT_READBACK Register Format for Command CMD_GFRC_READ_CORE



C.45 CMD_GFRC_READ_HALT

Field	Value
CONNECT_CMD[7:0]	0x49

Each core or the external master can use the CMD_GFRC_READ_HALT command to read the status of GFRC.

The CONNECT_READBACK register returns the value of internal HALT register indicating whether GFRC is halted or not.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

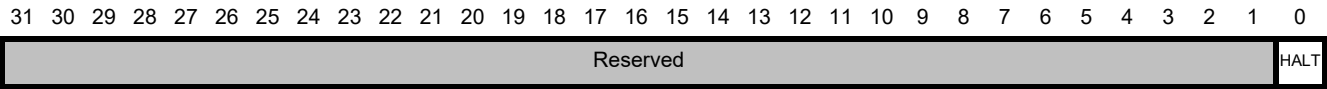
CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal HALT register.

The halt bit is set when all cores specified in the internal CORE register (by CMD_GFRC_SET_CORE command) have been halted. If any one of the core specified in the internal CORE register starts running, the GFRC will start running and the halt bit is cleared.

Figure C-24 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command.

Figure C-27 CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command



C.46 CMD_GFRC_CLK_ENABLE

Field	Value
CONNECT_CMD[7:0]	0x4A

Each core or external master can use the CMD_GFRC_CLK_ENABLE command to enable GFRC clock.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.47 CMD_GFRC_CLK_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x4B

Each core or external master can use the CMD_GFRC_CLK_DISABLE command to disable GFRC clock.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.48 CMD_GFRC_READ_CLK_STATUS

Field	Value
CONNECT_CMD[7:0]	0x4C

Each core or external master can use CMD_GFRC_READ_CLK_STATUS command to read the internal clock status register of the GFRC unit.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the content of internal clock status register after CMD_GFRC_READ_CLK_STATUS command.

Figure C-28 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command.

Figure C-28 CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																															S

When read result value is 0, it indicates that current GFRC clock is running.

When read result value is 1, it indicates that current GFRC clock is gated off.

C.49 CMD_PDM_SET_PM

Field	Value
CONNECT_CMD[7:0]	0x81

Each core or external master can use the CMD_PDM_SET_PM command to set the power mode for specific ARConnect group.



PDM ignores the command to power-down the power group if it has already been in power down mode.

Parameter Format

Parameter indicates group:

- 0x1: to IPC Group

CONNECT_WDATA Register Format

Figure C-29 CONNECT_WDAT Register Format for CMD_PDM_SET_PM Command

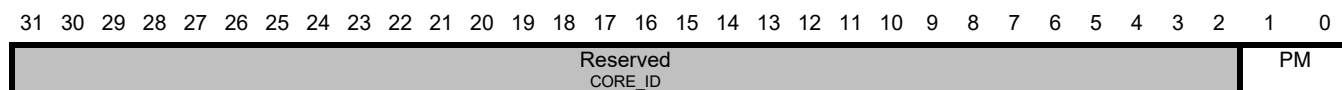


Table C-11 CONNECT_WDATA Register Field Descriptions for CMD_PDM_SET_PM

Field	Bits	Description
PM	[1:0]	Indicate power mode ■ 0x0: Power on mode ■ 0x1: Power down mode 1 ■ 0x2: Power down mode 2 ■ 0x3: Reserved

CONNECT_READBACK Register Format

N/A

C.50 CMD_PDM_READ_PSTATUS

Field	Value
CONNECT_CMD[7:0]	0x82

Each ARConnect group is assigned a PSTATUS register to record its previous power mode, current power mode, power up cause.

Each core or external master can use the CMD_PDM_READ_PSTATUS command to read the internal PM register of the specified ARConnect group.

Parameter Format

Parameter indicates group:

- 0x1: to IPC Group

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns data read out from the PM register of the specified component.

Figure C-30 CONNECT_READBACK Register Format for CMD_PDM_READ_STATUS Command

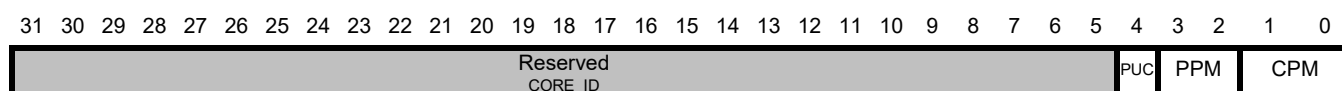


Table C-12 CONNECT_READBACK Register Field Descriptions for CMD_PDM_READ_STATUS

Field	Bits	Description
CPM	[1:0]	Current power mode of IPC group. ■ 0x0: Power on mode ■ 0x1: Power down PM1 mode ■ 0x2: Power down PM2 mode ■ 0x3: Reserved
PPM	[3:2]	Previous power mode of IPC group: ■ 0x0: Power on mode ■ 0x1: Power down PM1 mode ■ 0x2: Power down PM2 mode ■ 0x3: Reserved
PUC	[4]	Power up cause.

C.51 CMD_IVC_SET

Field	Value
CONNECT_CMD[7:0]	0x91

This command is used to program the internal INTVBASE register of specified ARC cores. Each core or the external master can use the `CMD_IVC_SET` command to set the hard coded `intvbase_in` value of specified ARC cores.

Each core is assigned an internal INTVBASE register in IVC. The `CONNECT_WDATA` register contains the data to be written into the internal INTVBASE register.

**Note**

When specifying the `CORE_ID` to the core not configured with `intvbase_ext` option, the value of `CONNECT_READBACK` register after `CMD_IVT_READ` command always returns zero.

Parameter Format

The parameter [4:0] field specifies the `CORE_ID` of ARC core whose internal `INTVBASE` register is updated with `CONNECT_WDATA` content.

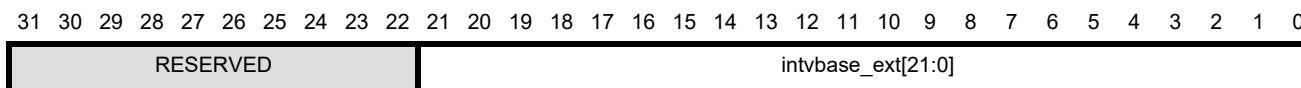
Besides, the bit [8] of the parameter field defines whether the content of `CONNECT_WDATA` updates the internal `INTVBASE` register of all cores.

- 1: update internal `INTVBASE` register of all ARC cores.
- 0: update the internal `INTVBASE` register of specified ARC core whose `CORE_ID` is specified by parameter[4:0].

CONNECT_WDATA Register Format

The data to be written into the `CONNECT_WDATA` register before the `CMD_IVC_SET` command is shown in the following figure. The valid bits are 22-bit.

Figure C-31 CONNECT_WDATA Register Format for CMD_IVC_SET Command

**Note**

When a core is not configured with the `intvbase_ext` option, the `CMD_IVC_SET` command is ignored.

CONNECT_READBACK Register Format

N/A

C.52 CMD_IVC_READ

Field	Value
CONNECT_CMD[7:0]	0x92

This command is used to read the internal `INTVBASE` register of specified core. Each core or the external master can use the `CMD_IVC_READ` command to read the hard coded `intvbase_in` value of specified ARC cores.

**Note**

When a core is not configured with the `intvbase_ext` option, the value of `CONNECT_READBACK` register after `CMD_IVT_READ` command always returns zero.

Parameter Format

The parameter [4:0] field specifies the CORE_ID of ARC core.

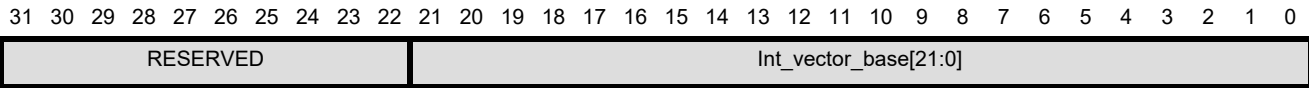
CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the internal INTVBASE register of specified core.

Figure C-32 CONNECT_READBACK Register Format for CMD_IVC_READ Command



C.53 CMD_PMU_SET_PUCNT

Field	Value
CONNECT_CMD[7:0]	0x51

An internal power-up duration counter PUCNT is implemented in the power management unit to controls the cycles from power switched on to isolation de-asserted. Each core or external master can use the CMD_PMU_SET_PUCNT command to set the PUCNT.

 **Note**

If the PUCNT is set to N, the duration will apply (256*N+8) cycles after each domain is powered up

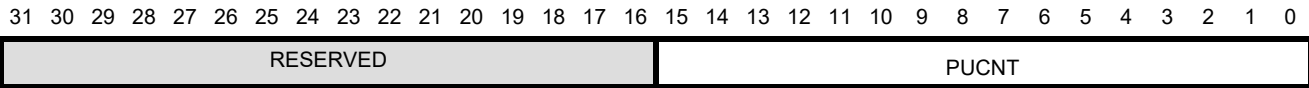
The CONNECT_WDATA register is used to set the value of this power-up duration counter.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure C-33 CONNECT_READBACK Register Format for CMD_PMU_SET_PUCNT Command



CONNECT_READBACK Register Format

N/A

C.54 CMD_PMU_READ_PUCNT

Field	Value
CONNECT_CMD[7:0]	0x52

Each core or external master can use the CMD_PMU_READ_PUCNT command to read the value of power-up duration counter in the PMU unit.

The CONNECT_READBACK register returns the content of power-up duration register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The layout of CONNECT_READBACK register is same as the CONNECT_WDATA layout of CMD_PMU_SET_PUCNT command.

C.55 CMD_PMU_SET_RSTCNT

Field	Value
CONNECT_CMD[7:0]	0x53

An internal reset delay counter RSTCNT is implemented in the power management unit to extend the reset cycles from isolation de-asserted to reset released. Each core or external master can use the CMD_PMU_SET_RSTCNT command to set the RSTCNT.



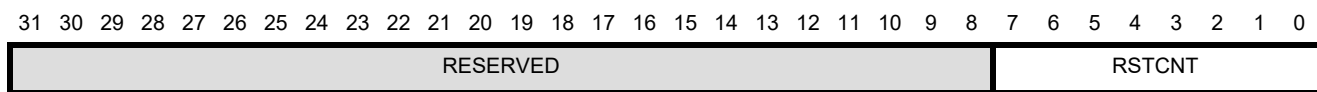
Note

If the RSTCNT is set to N, the reset duration will extend $(8*N+8)$ cycles after the isolation control is de-asserted.

The CONNECT_WDATA register is used to set the value of this reset duration counter.

Parameter Format

N/A

CONNECT_WDATA Register Format**Figure C-34 CONNECT_READBACK Register Format for CMD_PMU_SET_RSTCNT Command****CONNECT_READBACK Register Format**

N/A

C.56 CMD_PMU_READ_RSTCNT

Field	Value
CONNECT_CMD[7:0]	0x54

Each core or external master can use the CMD_PMU_READ_RSTCNT command to read the value of reset duration counter in the PMU unit.

The CONNECT_READBACK register returns the content of reset duration register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The layout of CONNECT_READBACK register is same as the CONNECT_WDATA layout of CMD_PMU_SET_RSTCNT command.

C.57 CMD_PMU_SET_PDCNT

Field	Value
CONNECT_CMD[7:0]	0x55

An internal power-down duration counter PDCNT is implemented in the power management unit to control the cycles from isolation asserted to power switched off. Each core or external master can use the CMD_PMU_SET_PDCNT command to set the PDCNT.

 **Note**

If the PDCNT is set to N, the duration will apply (256*N+8) cycles after the isolation control is asserted. Then the power will be switched off.

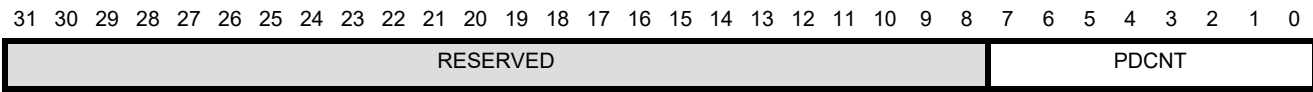
The CONNECT_WDATA register is used to set the value of this power-down duration counter.

Parameter Format

N/A

CONNECT_WDATA Register Format

Figure C-35 CONNECT_READBACK Register Format for CMD_PMU_SET_PDCNT Command



CONNECT_READBACK Register Format

N/A

C.58 CMD_PMU_READ_PDCNT

Field	Value
CONNECT_CMD[7:0]	0x56

Each core or external master can use the CMD_PMU_READ_PDCNT command to read the value of power-down duration counter in the PMU unit.

The CONNECT_READBACK register returns the content of power-down duration register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The layout of CONNECT_READBACK register is same as the CONNECT_WDATA layout of CMD_PMU_SET_PDCNT command.

C.59 CMD_IDU_ENABLE

Field	Value
CONNECT_CMD[7:0]	0x71

IDU has a global `ENABLE` register to disable or enable the common interrupts. The global `ENABLE` register is zero after reset.

Each core or the external master can use the `CMD_IDU_ENABLE` command to enable the common interrupts using this `ENABLE` register.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.60 CMD_IDU_DISABLE

Field	Value
CONNECT_CMD[7:0]	0x72

Each core or an external master can use the `CMD_IDU_DISABLE` command to disable IDU.

Parameter Format

N/A

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.61 CMD_IDU_READ_ENABLE

Field	Value
CONNECT_CMD[7:0]	0x73

Each core can use the `CMD_IDU_READ_ENABLE` command to read the internal `ENABLE` register of IDU.

Parameter Format

N/A

CONNECT_WDATA Register Format

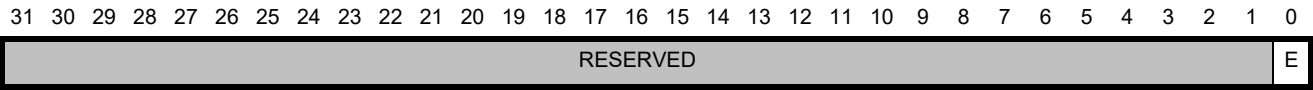
N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the contents of the internal `ENABLE` register after the `CMD_IDU_READ_ENABLE` command.

The layout of the `CONNECT_READBACK` register is shown in [Figure C-36](#). A value of 1 indicates that IDU is enabled, otherwise it is disabled.

Figure C-36 `CONNECT_READBACK` Register Format for `CMD_IDU_READ_ENABLE` Command



C.62 CMD_IDU_SET_MODE

Field	Value
CONNECT_CMD[7:0]	0x74

Each common interrupt is assigned an internal `MODE` register in IDU. Each core can use the `CMD_IDU_SET_MODE` command to set the triggering mode and distribution mode (in the internal `MODE` register) for the specified common interrupt. The `PARAMETER` field is used to specify the index of common interrupt. The `CONNECT_WDATA` register contains the mode value to be set into the `MODE` register.

The default value of the `MODE` register value is `0x0`.

Parameter Format

The `Parameter [6:0]` field defines the index of a common interrupt, for example `0x0` indicates the common interrupt 0; `0x8` indicates the common interrupt 8.

CONNECT_WDATA Register Format

Figure C-37 CONNECT_WDATA Register Format for CMD_IDU_SET_MODE Command

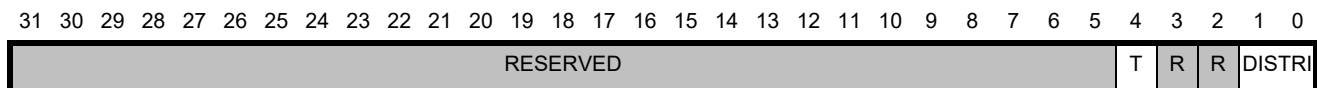


Table C-13 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MODE

Field	Bits	Description
DISTRI	[1:0]	The distribution mode: 0x0: Round-robin mode (default) 0x1: First-acknowledge mode 0x2: All-destination mode
T	[4]	The trigger mode: 0x0: level triggered (default) 0x1: edge triggered NOTE: This field is invalid for common interrupts programmed by the software.

CONNECT_READBACK Register Format

N/A

C.63 CMD_IDU_READ_MODE

Field	Value
CONNECT_CMD[7:0]	0x75

Each core can use the `CMD_IDU_READ_MODE` command to read the triggering mode and distribution mode using the internal `MODE` register for the specified common interrupt. The `PARAMETER` field is used to specify the common interrupt.

The `CONNECT_READBACK` register returns the mode value for the specified common interrupt.

Parameter Format

The `Parameter [6:0]` field defines the index of a common interrupt.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The layout of it is same as the `CONNECT_WDATA` register of the `CMD_IDU_SET_MODE` command.

C.64 CMD_IDU_SET_DEST

Field	Value
CONNECT_CMD[7:0]	0x76

Each common interrupt is assigned a DEST register in IDU. Each core can use the CMD_IDU_SET_DEST command to set the target cores (using the internal register DEST register) to receive the specified common interrupt when it is triggered. The default value of the DEST register value is 0x0.

The PARAMETER field is used to specify the index of a common interrupt. The CONNECT_WDATA register contains the destination value to be set in the DEST register.

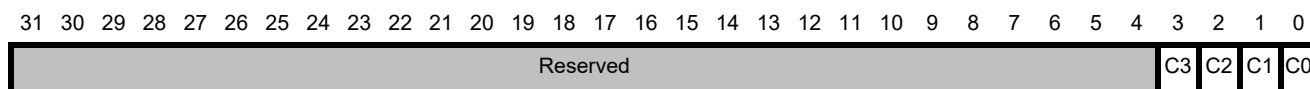
Parameter Format

The Parameter [6:0] field defines the index of a common interrupt.

CONNECT_WDATA Register Format

For a four-core system, the layout of the CONNECT_WDATA register is shown in [Figure C-38](#).

Figure C-38 CONNECT_WDATA Register Format for CMD_IDU_SET_DEST Command



The field descriptions are shown in [Table C-14](#).

When programmed in the all-destination distribution mode, for level-triggered common interrupts triggered by the external system, it is invalid to set more than one target core in its DEST register. Nevertheless, software can write multi-hot bits to the DEST register (for level-triggered interrupts) although it is invalid, and the DEST can be read back with this multi-hot bit value. But IDU distributes this interrupt to the first valid target core with the smallest CORE_ID specified in the DEST register.

Table C-14 CONNECT_WDATA Register Field Description for CMD_IDU_SET_DEST

Field	Bits	Description
C0	[0]	Indicates whether Core0 is selected for the specified common interrupt 0 = Not targeted 1 = Core0 is targeted
C1	[1]	Indicates whether Core1 is selected for the specified common interrupt 0 = Not targeted 1 = Core1 is targeted
C2	[2]	Indicates whether Core2 is selected for the specified common interrupt 0 = Not targeted 1 = Core 2 is targeted

Table C-14 CONNECT_WDATA Register Field Description for CMD_IDU_SET_DEST

Field	Bits	Description
C3	[3]	Indicates whether Core3 is targeted for the specified common interrupt 0 = Not targeted 1 = Core3 is targeted

CONNECT_READBACK Register Format

N/A

C.65 CMD_IDU_READ_DEST

Field	Value
CONNECT_CMD[7:0]	0x77

Each core can use the `CMD_IDU_READ_DEST` command to read the internal `DEST` register of the specified common interrupt. The `PARAMETER` field is used to specify the common interrupt.

The `CONNECT_READBACK` register returns the value of the `DEST` register of the specified common interrupt.

Parameter Format

The `Parameter [6:0]` field defines the index of the common interrupt.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the content of the internal `DEST` register of the specified common interrupt after the `CMD_IDU_READ_DEST` command, the layout of which is the same as the `CONNECT_WDATA` register of the `CMD_IDU_SET_DEST` command.

C.66 CMD_IDU_GEN_CIRQ

Field	Value
CONNECT_CMD[7:0]	0x78

Each core or the external master can use the `CMD_IDU_GEN_CIRQ` command to assert the specified common interrupt. This interrupt is referred to as a software generated interrupt.

The `PARAMETER` field is used to specify the common interrupt.

Parameter Format

The `Parameter [6:0]` field specifies the index of the common interrupt.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.67 CMD_IDU_ACK_CIRQ

Field	Value
CONNECT_CMD[7:0]	0x79

Each core can use the `CMD_IDU_ACK_CIRQ` command to acknowledge and clear the common interrupt. The `PARAMETER` field is used to specify the index of the common interrupt.



Note

If the external master sends the `CMD_IDU_ACK_CIRQ` command to IDU, the command is treated as a NOP.

If the external system common interrupt is edge-triggered or a common interrupt is programmed by software, the pending bits corresponding to the receiving cores are set after detecting the rising edge of external common interrupt line; the pending bit corresponding to each receiving core is cleared after receiving the `CMD_IDU_ACK_CIRQ` command from the corresponding receiving core.

If the external system common interrupt is level-triggered, the pending bits corresponding to the receiving core is set whenever the external common interrupt line is asserted, and cleared after the de-asserting of the external common interrupt line.



Note

If the external system common interrupt mode is level-triggered, there is no need for the receiving core to clear the common interrupt by using this command. Instead, the core should clear the common interrupt line in the external system.

IDU treats the `CMD_IDU_ACK_CIRQ` command as a NOP if it is to a pending external system level-triggered interrupt or to a non-pending interrupt.

If the common interrupt is triggered by the `CMD_IDU_GEN_CIRQ` command, the pending bits corresponding to each receiving core is cleared by the `CMD_IDU_ACK_CIRQ` command.



Note

If the pending bit is cleared by the `CMD_IDU_ACK_CIRQ` command and the new interrupt is registered simultaneously, the pending bit is cleared.

The interrupt lines to core is rescinded along with the corresponding pending bit to this core is cleared.

The Parameter field is used to specify the common interrupt.

Parameter Format

The `Parameter [6:0]` field specifies the index of the common interrupt.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

N/A

C.68 CMD_IDU_CHECK_STATUS

Field	Value
CONNECT_CMD[7:0]	0x7A

Each common interrupt is assigned an internal `STATUS` register indicating its pending status in each processor core of the system. The default value of the `STATUS` register is 0x0.

Each core can use the `CMD_IDU_CHECK_STATUS` command to read the internal `STATUS` register of the specified common interrupt.

The `PARAMETER` field is used to specify the common interrupt. The `CONNECT_READBACK` register returns the internal `STATUS` register of the specified common interrupt.

Parameter Format

The `Parameter [6:0]` field defines the index of the common interrupt.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

For a four-core system, the `CONNECT_READBACK` register returns the content of the internal `STATUS` register of the specified common interrupt after the `CMD_IDU_CHECK_STATUS` command, as shown in [Figure C-39](#).

Figure C-39 CONNECT_READBACK Register Format for CMD_IDU_CHECK_STATUS Command

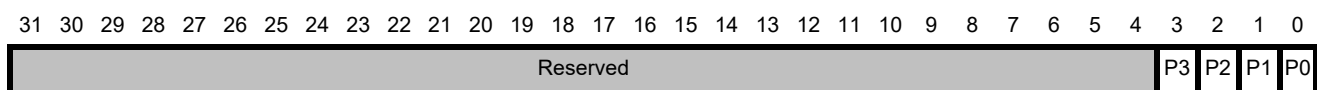


Table C-15 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_STATUS

Field	Bits	Description
P0	[0]	Pending status to Core0 0 = there is no pending interrupt or a former interrupt has been cleared 1 = there is a pending interrupt
P1	[1]	Pending status to Core1 0 = there is no pending interrupt or a former interrupt has been cleared 1 = there is an pending interrupt
P2	[2]	Pending status to Core2 0 = there is no pending interrupt or a former interrupt has been cleared 1 = there is an pending interrupt
P3	[3]	Pending status to Core3 0 = there is no pending interrupt or a former interrupt has been cleared 1 = there is an pending interrupt

**Note**

The pending bits reflect the interrupt status. If the pending bits were asserted and not cleared, and the same interrupt is re-generated again (referred as the second interrupt), the pending bits are unchanged, which means the 2nd interrupt is unseen. So ensure that the same interrupt is not regenerated before the first interrupt is cleared, otherwise the result is unpredictable because the second interrupt may be lost.

C.69 CMD_IDU_CHECK_SOURCE

Field	Value
CONNECT_CMD[7:0]	0x7B

Each common interrupt is assigned an internal **SOURCE** register indicating the source information of the specified common interrupt. The default value of the **SOURCE** register is 0x0.

Each core can use the **CMD_IDU_CHECK_SOURCE** command to read the internal **SOURCE** register of the specified common interrupt.

The **PARAMETER** field is used to specify the common interrupt.

The **CONNECT_READBACK** register returns the internal **SOURCE** register of the specified common interrupt.

Parameter Format

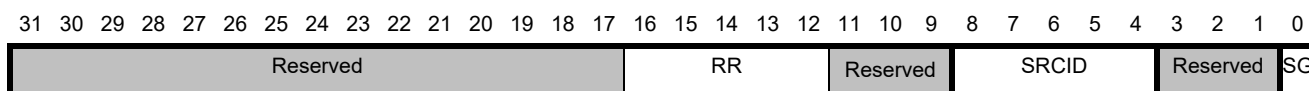
The **Parameter [6:0]** field defines the index of the common interrupt.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the content of the internal `SOURCE` register of the specified common interrupt after the `CMD_IDU_CHECK_SOURCE` command, as shown in [Figure C-40](#).

Figure C-40 CONNECT_READBACK Register Format for CMD_IDU_CHECK_SOURCE command**Table C-16 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_SOURCE**

Field	Bits	Description
SG	[0]	Software generation flag bit 0 = the interrupt is not generated by software by using the <code>CMD_IDU_GEN_CIRQ</code> command. 1 = the interrupt is generated by software by using the <code>CMD_IDU_GEN_CIRQ</code> command. NOTE: The SG bit is cleared by the <code>CMD_IDU_ACK_CIRQ</code> command. NOTE: The SG bit is not set in following two conditions: - There is a pending interrupt generated by external common interrupt line. - An external common interrupt line is asserted concurrently.
SRCID	[8:4]	Indicates <code>CORE_ID</code> of the core generated the software interrupt. This field is only valid when the SG bit is 1. Core0 is indicated by 0x0, Core1 by 0x1, Core 2 by 0x2 and so on. If the ASI is present, the external master is indicated by the last core ID +1 (so it will be 0x2 for a two-core system and 0x4 for a four-core system). When the SG bit is 1, the SRCID can be overridden with another <code>CORE_ID</code> if that core sends the <code>CMD_IDU_GEN_CIRQ</code> to assert the interrupt again.
RR (Round-Robin)	[16:12]	Indicates the round-robin counter pointed core that has received the interrupt in round-robin mode when it is triggered. This field is only valid for the interrupt whose distribution mode is set as round-robin policy. Core0 is indicated by 0x0, core1 is indicated by 0x1, and so on. The round-robin counter is cleared to zero after reset, or set to the first core (specified in the DEST register) when this interrupt's distribution mode is programmed as round-robin or the value of the DEST register is changed. The round-robin counter is incremented to next targeted core (specified in the DEST register) every time this interrupt is cleared.

C.70 CMD_IDU_SET_MASK

Field	Value
CONNECT_CMD[7:0]	0x7C

Each common interrupt is assigned an internal MASK register in the IDU. The default value of this MASK register is 0x0. Each core can use the CMD_IDU_SET_MASK command to mask or unmask the specified common interrupt using this MASK register.

The PARAMETER field is used to specify the index of common interrupt.

If the interrupt is generated by the external system or software (by CMD_IDU_GEN_CIRQ command), the following principles apply to the masking and unmasking operations:

If an interrupt is already seen by the IDU when the masking operation is taking effect, for either an edge triggered, level triggered interrupt, and software programming interrupt; the “masking” operation does not impact the internal states

If an interrupt is not yet seen by the IDU when the masking operation is taking effect. For either an edge-triggered, level-triggered interrupt, and software programming interrupt; IDU does not see this interrupt.

1. The edge trigger interrupt is lost.
2. The level interrupt is completely blocked (the clearing or asserting of this interrupt has no impact the interrupt distribution unit internal states). But if the interrupt is still asserting when this interrupt is unmasked, the consequence is same to the asserting of this interrupt.
 - a. The PEND bit in status register is set, and the interrupt line to corresponding core i asserted.
3. The software programming interrupt is lost.



Note

It is strongly recommended to make interrupt origination inactive before re-programing the MASK register of each common interrupt (the CMD_IDU_SET_MASK command). Otherwise, the result is unpredictable because the CMD_IDU_SET_MASK command may take effect before or after IDU has seen the interrupt.

Parameter Format

The Parameter [6:0] field defines the index of a common interrupt.

CONNECT_WDATA Register Format

Figure C-41 CONNECT_WDATA Register Format for CMD_IDU_SET_MASK command

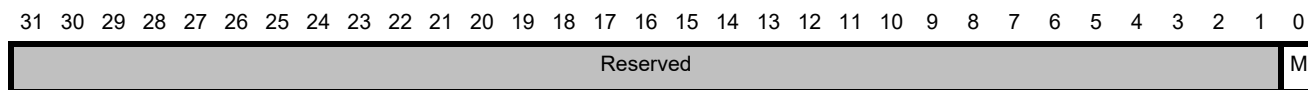


Table C-17 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MASK

Field	Bits	Description
M	[0]	0: Unmask this interrupt (default) 1: Mask this interrupt

CONNECT_READBACK Register Format

N/A

C.71 CMD_IDU_READ_MASK

Field	Value
CONNECT_CMD[7:0]	0x7D

Each core or the external master can use the `CMD_IDU_READ_MASK` command to read the internal MASK register of the specified common interrupt.

The `PARAMETER` field is used to specify the common interrupt.

The `CONNECT_READBACK` register returns the mode value for the specified common interrupt.

Parameter Format

The `Parameter [6:0]` field defines the index of the common interrupt.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the content of the internal MASK register of the specified common interrupt after the `CMD_IDU_READ_MASK` command, the layout of which is the same as the `CONNECT_WDATA` register of the `CMD_IDU_SET_MASK` command.

C.72 CMD_IDU_CHECK_FIRST

Field	Value
CONNECT_CMD[7:0]	0x7E

Each core can use the `CMD_IDU_CHECK_FIRST` command to check if it is the first-acknowledging core to the common interrupt; if IDU is programmed in the first-acknowledged mode.

 **Note**

If the external master sends the `CMD_IDU_CHECK_FIRST` command to IDU through ASI, it is treated as a NOP.

The `PARAMETER` field is used to specify the index of the common interrupt.

The `CONNECT_READBACK` register returns the value to indicate if it is the first-acknowledging core.

Parameter Format

The `Parameter [6:0]` field defines the index of a common interrupt.

CONNECT_WDATA Register Format

N/A

CONNECT_READBACK Register Format

The `CONNECT_READBACK` register returns the value to indicate if it is the first-acknowledging core, as shown in [Figure C-42](#).

Figure C-42 CONNECT_READBACK Register Format for CMD_IDU_CHECK_FIRST Command

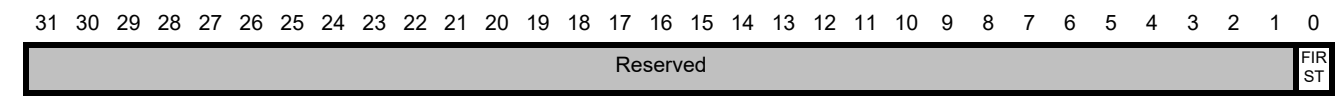


Table C-18 CONNECT_WDATA Register Field Description for CMD_IDU_CHECK_FIRST

Field	Bits	Description
FIRST	[0]	0 = this core is not the first-acknowledging core, or this interrupt is not programmed as a first-acknowledging mode, or there is no pending status for this interrupt. 1 = this core is the first-acknowledging core, and this interrupt is programmed as a first-acknowledging mode, and this interrupt is pending.

C.73 Examples: ARConnect ICI

C.73.1 Scenario 1

Core1 generates an inter-core interrupt to Core2. Core2 receives and services the interrupt, and then generates an interrupt acknowledgment to Core1.

In this scenario, Core1 is the interrupt initiator core. To avoid overriding the previous interrupt, Core1 should check the interrupt status before generating a new interrupt to Core2. A segment program is shown in [Example C-1](#).

Example C-1 Segment Program of Scenario 1 — Core1

<code>chk status:</code>	Check the interrupt status of Core2, so that any previous interrupt is not overridden.
<code>SR 0x00000203, [CONNECT_CMD]</code>	Send the <code>CMD_INTRPT_READ_STATUS</code> command to check the status of the interrupt it generated to Core2.
<code>LR r1, [CONNECT_READBACK] CMP r1, 1</code>	The interrupt status is read back using the <code>CONNECT_READBACK</code> register.
<code>BEQ chk_status</code>	Exit the loop until the corresponding status bit is cleared.
<code>SR 0x00000201, [CONNECT_CMD]</code>	Send the <code>CMD_INTRPT_GENERATE_IRQ</code> command to generate inter-core interrupt to Core2.

Core2 is the interrupt receiver core. A program segment of its interrupt handler is shown in [Example C-2](#).

Example C-2 Segment Program of Scenario 1 — Core2

<code>SR r2, [CONNECT_CMD]</code>	The content of r2 is 0x00000004, indicating that the <code>CMD_INTRPT_CHECK_SOURCE</code> command is being sent to check the source of the interrupt.
<code>LR r4, [CONNECT_READBACK]</code>	Read back the vector value from the <code>CONNECT_READBACK</code> register to core register r4.
<code>AND.F 0, r4, 0x00000002 BEQ gen_ack_to_c1</code>	Check whether the bit corresponding to Core1 is asserted.

```
gen_ack_to_c1:
SR r5, [CONNECT_CMD]
```

The content of r5 is 0x00000102, indicating that the `CMD_INTRPT_GENERATE_ACK` command is issued to acknowledge the interrupt from Core1.

Note: Make sure the interrupt has been de-asserted before exiting the interrupt handler, for example, by using `CMD_INTRPT_CHECK_SOURCE` command again in a loop until the initiator core has de-asserted the interrupt.

C.73.2 Scenario2

Core0, Core1, and Core2 generate inter-core interrupts to Core3. Core3 services the interrupts and generates interrupt acknowledgments to Core0, Core1, and Core2 respectively.

In this scenario, Core0, Core1, and Core2 are the interrupt initiators. The segment program of each core can be the same as scenario1.

Example C-3 Segment Program of Scenario 2 - Core0, Core1, Core2

```
chk_status:                                Check the interrupt status of Core3, so that any previous interrupt is
                                           not overridden.

SR 0x00000303,                             Send the CMD_INTRPT_READ_STATUS command.
[CONNECT_CMD]

LR r1, [CONNECT_READBACK]                 The interrupt status is read back from the CONNECT_READBACK
CMP.F r1, 0x00000001                      register.

BEQ    chk_status                          Exit the loop until the corresponding status bit is cleared.

SR 0x00000301,                             Send the CMD_INTRPT_GENERATE_IRQ command to Core3 to
[CONNECT_CMD]                             generate inter-core interrupt.

...
```

Core3 is the interrupt receiver; a segment program of its interrupt handler is shown in [Example C-4](#).

Example C-4 Segment Program of Scenario 2 - Core3

```
serve_irq:

SR r0, [CONNECT_CMD]                     The content of r0 is 0x00000040, which indicates that the
                                           CMD_INTRPT_CHECK_SOURCE command is being sent to
                                           check the sources of the generated interrupts.

LR r1, [CONNECT_READBACK]                 The vector value is read back from the CONNECT_READBACK
                                           register to core register r1.

CMP r1, 0                                Check whether all interrupts have been serviced.
```

BEQ end_serve_irq	
...	
MOV r2, 1	Set the start value 1 for index control.
serve_irq_for_each_core:	
MOV r3, r1	Move the interrupt source vector to temporary register r3.
SUB r10, r2, 1	Generate the <code>shift_loop</code> control value.
shift_loop:	
CMP r10, 0	
BEQ gen_ack_to_each_core	If equal, go to generate interrupt acknowledges.
LSR r3, r3	Shift right operation of interrupt source vector.
SUB r10, r10, 1	
BAL shift_loop	
gen_ack_to_each_core:	
AND.F 0, r3, 0x00000001	Check the value of bit0 for shifted interrupt source vector.
BEQ incr_index	If the AND result is zero, do nothing and increment the index; otherwise, go on, generate interrupt, and acknowledge to the core with a specific index value.
...	
SR r11, [CONNECT_CMD]	Instructions to generate the command value and store it into r11. In this case, the content of r11 is sent using the command <code>CMD_INTRPT_GENERATE_ACK</code> to Core0, Core1, and Core2 respectively.
...	
incr_index:	
ADD r2, r2, 1	
CMP r2, CONNECT_CORE_NUMBER	
BLS serve_irq_for_each_core	If the index value is less than or equal to the number of cores in the system, go on to check the interrupt status.
end_serve_irq:	
RTIE	Note: Ensure that the corresponding bit of the interrupt source has been cleared before exiting the handler.

C.74 Examples: ARConnect Semaphore Unit

C.74.1 Scenario 1

In this scenario, Core1 claims the ownership of semaphore 8. After some processing, Core1 releases semaphore 8.

Example C-5 Segment Program of Scenario 1- Core1

claim: SR 0x00000811, [CONNECT_CMD]	Core1 sends the CMD_SEMA_CLAIM_AND_READ command to claim ownership of semaphore 8.
LR r0, [CONNECT_READBACK]	The claim result is read back from the CONNECT_READBACK register.
CMP r0, 1	Check whether the claim is successful.
BNE claim	If the claim is a failure, go back to claim semaphore 8. Otherwise, continue with the claim.
SR 0x00000812, [CONENCT_CMD]	Send the CMD_SEMA_RELEASE command to release semaphore 8 and allow it to be available for other cores.

C.74.2 Scenario 2

Core2 claims the ownership of semaphore 8, that is already claimed and owned by other cores. Core2 compulsively releases semaphore 8, and then claims its ownership.

Example C-6 Segment Program of Scenario 2 - Core 2

claim: SR 0x00000811, [CONNECT_CMD]	Core2 sends the CMD_SEMA_CLAIM_AND_READ command to claim the ownership of semaphore 8.
LR r0, [CONNECT_READBACK]	The claim result is read from the CONNECT_READBACK to core register r0.
CMP r0, 1	Check whether the claim is successful.
BNE force_release	If the claim is a failure, force release semaphore 8. Otherwise, continue with the claim.
force_release: SR 0x00000813, [CONNECT_CMD]	Core2 sends the CMD_SEMA_FORCE_RELEASE command to compulsively release semaphore 8 and allow it to be available.
BA claim ...	

C.75 Examples: ICM

C.75.1 Scenario 1

Core1 is the message producer and Core2 is the message consumer. Several messages are to be passed between Core1 and Core2.

Messages are written into and read from the message-passing SRAM. Address 0x00 in the message passing SRAM is used to transfer the communication flag between Core1 and Core2.

Example C-7 Segment Program of Scenario 1 - Core1

SR 0x064, [CONNECT_WDATA]	Set the message passing SRAM address to be 0x064.
SR 0x00000021, [CONNECT_CMD]	Send the CMD_MSG_SRAM_SET_ADDR command to ARConnect.
SR -4, [CONNECT_WDATA]	Set the SRAM address offset value to -4 (a negative value).
SR 0x00000023, [CONNECT_CMD]	Send the CMD_MSG_SRAM_SET_ADDR_OFFSET command.
...	
SR r1 [CONNECT_WDATA]	r1 contains the data to be written into message passing SRAM.
SR 0x00000026, [CONNECT_CMD]	Send the CMD_MSG_SRAM_WRITE_INC command. The data in the CONNECT_WDATA register is written into address 0x064. After that, the SRAM address increments to 0x060 (current address + offset value).
SR r2, [CONNECT_WDATA]	r2 contains the data to be written into the SRAM.
SR 0x00000026, [CONNECT_CMD]	Send the CMD_MSG_SRAM_WRITE_INC command. The data in the CONNECT_WDATA register is written into address 0x060. After that, the SRAM address decrements to 0x05c.
SR 0x55555555, [CONNECT_WDATA]	After producing the message, set address 0x00 to 0x55555555 to tell Core2 that the message is ready.
SR 0x00000027, [CONNECT_CMD]	Send the CMD_MSG_SRAM_WRITE_IMM command. The SRAM address is specified in the parameter field (0x00).

C.76 Examples: ICD

C.76.1 Conditionally Halt Cores Scenario

Core0 is halted by a breakpoint during the execution. The debugger can program ICD to trigger a conditional halt to other cores when a breakpoint halt occurs in Core0.

[Example C-8](#) shows a program segment of Core2 if the debugger programs ICD through Core2.

Example C-8 Conditionally Halt Cores Scenario

```
SR 0x0000000e,  
[CONNECT_WDATA]
```

Core1, Core2, and Core3 are selected for the following debug command.

```
SR 0x00000036,  
[CONNECT_CMD]
```

Send the `CMD_DEBUG_SET_SELECT` command to the ICD, the cores selected by the previous SR instruction are halted by the global halt pulse.

```
SR 0x00000001,  
[CONNECT_WDATA]
```

Core0 is selected for the following debug commands.

```
SR 0x00000434,  
[CONNECT_CMD]
```

Send the `CMD_DEBUG_SET_MASK` command to ICD. Parameter[7:0] is 4'b0100 to indicate the mask value of SH, BH, AH, and H bits to control the generation of the global halt pulse.

C.76.2 Selectively Halt Cores Scenario

Core0 programs ICD to selectively halt other cores in the system.

[Example C-9](#) shows a program segment of Core0.

Example C-9 Selectively Halt Cores Scenario

```
SR 0x0000000e,  
[CONNECT_WDATA]
```

Core1, Core2, and Core3 are selected for the following debug commands.

```
SR 0x00000032,  
[CONNECT_CMD]
```

Send the `CMD_DEBUG_HALT` command to ICD. The cores selected by the previous SR instruction are halted.

C.76.3 Selectively Run Cores Scenario

All cores have already halted. The debugger programs ICD to run all cores in the system.

Example C-10 Selectively Run Cores Scenario

```
SR 0x0000000f,  
[CONNECT_WDATA]
```

Core0 to 3 are selected for following debug command.

```
SR 0x00000033, [CONNECT_CMD]
```

Send the `CMD_DEBUG_RUN` command to ICD. The cores selected by the previous SR instruction start to run.

C.76.4 Selectively Reset Cores Scenario

Core0 is the master CPU of a two core system.

[Example C-11](#) shows a program segment of Core0.

Example C-11 Selectively Reset Cores Scenario

wait_core1_halt:

SR 0x00000038, [CONNECT_CMD]	Send the CMD_DEBUG_READ_EN command to ICD.
LR r0, [CONNECT_READBACK]	
AND.F 0, r0, 0x00000002	Check the EN bit of Core1.
BNZ wait_core1_halt	Reallocate the program of Core1.
SR 0x00000002, [CONNECT_WDATA]	Core1 is selected for the following debug command.
SR 0x00000031, [CONNECT_CMD]	Send the CMD_DEBUG_RESET command to reset Core1.

Example C-12 Segment Program of Scenario 1 - Core2

...	Other tasks are processed then waiting messages are passed from core1.
wait_flag: SR 0x0000002a, [CONNECT_CMD]	Send the CMD_MSG_SRAM_READ_IMM command.
LR r0, [CONNECT_READBACK]	The data is read back from the CONNECT_READBACK register.
CMP r0, 0x55555555	Check whether Core1 produced the message.
BNE wait_flag	If failed, go back to wait for the message.
SR 0x050, [CONNECT_WDATA]	Set the message passing SRAM address to be 0x050.
SR 0x00000021, [CONNECT_CMD]	Send the CMD_MSG_SRAM_SET_ADDR command.
SR 4, [CONNECT_WDATA]	Set the SRAM address offset value to 4 (a positive value).
SR 0x00000023, [CONNECT_CMD]	Send the CMD_MSG_SRAM_SET_ADDR_OFFSET command.
SR 0x00000029, [CONNECT_CMD]	Send the CMD_MSG_SRAM_READ_INC command to read content in address 0x00. The data is read to the CONNECT_READBACK register from the SRAM address 0x050. After that, the address increments to 0x054 (current address + offset value).
LR r1, [CONNECT_READBACK]	Data is read from the address 0x050 is stored to core register r1.
SR 0x00000029, [CONNECT_CMD]	Send the CMD_MSG_SRAM_READ_INC command. The data is read out to the CONNECT_READBACK register from address 0x054. After that, the SRAM address increments to 0x058.

LR r2, [CONNECT_READBACK]

...

Data read from the address 0x054 is stored to core register r2.

C.77 Examples: GFRC

Example C-13 Segment Program to Read Value of GFRC

SR 0x00000042, [CONNECT_CMD]

LR r5, [CONNECT_READBACK]

SR 0x00000043, [CONNECT_CMD]

LR r6, [CONNECT_READBACK]

Send the CMD_GRTC_READ_LO command.

The lower 32-bit value of GRTC are returned to the CONNECT_READBACK register, and stored to r5 register of the core.

Send the CMD_GRTC_READ_HI command.

The higher 32-bit value of GRTC is returned to the CONNECT_READBACK register, and store to r6 register of the core.

C.78 Examples: IDU

For a four-core system, the initiator core programs the IDU's common interrupt 1:

Distribution Mode	First-acknowledge mode
Triggering Mode	N/A
Destination Cores	Four cores
Mask Mode	Unmasked

After programming IDU, the initiator core generates common interrupt 1 by using the CMD_IDU_GEN_CIRQ command. Then the IDU asserts the interrupt lines to all cores specified in its DEST register.

[Example C-14](#) shows a segment program of the initiator core - Core0.

Example C-14 Segment Program of Common Interrupt 1 Generated by Core 0

<code>_reset:</code>	Reset the exception base
<code> .long _start</code>	
<code> ...</code>	
 <code>irq25: .long irq_idu_handler</code>	Common interrupt1 corresponds to IRQ25
 <code>test_start:</code>	On-reset start point of the program
 <code>.text</code>	
<code>.global _start</code>	
<code>_start:</code>	
<code> ...</code>	
 <code>SR 0x00000072, [CONNECT_CMD]</code>	Use the <code>CMD_IDU_DISABLE</code> command to disable the interrupt distribution unit
 <code>SR 0x00000011, [CONNECT_WDATA]</code>	Set the distribution mode of common interrupt 1.
 <code>SR 0x00000174, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_SET_MODE</code> command to ARConnect, parameter <code>[6:0] = 7'd01</code> indicates common interrupt1 is set
 <code>SR 0x0000000f, [CONNECT_WDATA]</code>	Four cores are targeted
 <code>SR 0x00000176, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_SET_DEST</code> command to ARConnect, parameter <code>[6:0] = 7'd01</code> indicates that the data in <code>CONNECT_WDATA</code> is written into the <code>DEST</code> register of common interrupt1
 <code>SR 0x00000000, [CONNECT_WDATA]</code>	Set the common interrupt MASK to 0
 <code>SR 0x0000017C, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_SET_MASK</code> command to ARConnect, parameter <code>[6:0] = 7'd01</code> to indicate that the common interrupt1 will be unmasked.
 <code>SR 0x00000071, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_ENABLE</code> command to enable the interrupt distribution unit
 <code>SR 0x00000178, [CONNECT_CMD]</code>	Send the <code>CMD_IDU_GEN_CIRQ</code> command to ARConnect, parameter <code>[6:0] = 7'd0</code> to indicate that the common interrupt1 is asserted.

C.79 Examples: IVC

The external master can access ARConnect IVC component through an ARConnect ASI interface.

1. Access MMIO CONNECT_WDATA register by writing with interrupt vector value for next run.
2. Access MMIO CONNECT_CMD register with CMD_IVC_SET command to specify the scope of ARC cores, that is, whether the written interrupt vector value is updated all ARC cores or a specified ARC core.
3. Reset the ARC cores and restart the program from newly programmed interrupt vector base.

Index

A

- ABS 380
- absolute instruction 380
- actionpoint
 - access requirements 1127
 - auxiliary registers 1125
 - control register 1133
 - extensions to debug register 1118, 1129
 - match mask register 1131
 - registers reset state 1128
 - system 1125
- actionpoint configuration register 1130
- active interrupts register 155
- ADC 382
- ADD 1381, 1387, 1389, 1391, 1393, 1467
- add instruction
 - syntax summary 273
- add with carry instruction 382
- add with scaled source instruction
 - shift 1 386
 - shift 2 388
 - shift 3 390
- ADD1 386
- ADD2 388
- ADD3 390
- address space 109
- addressing modes 121
 - compare and branch 123
 - conditional branch 123
 - conditional execution 122
 - null instruction format 122
- AE flag 149
- AEX 392
- AND 394
- and instruction 394
- AP_BUILD 1130
- ARCV2
 - introduction 69, 95
 - key features 95
- arithmetic shift left instruction 396
- arithmetic shift right instruction 401
 - multiple 403
- ASL 396
- ASL multiple 398
- ASR 401
- ASR multiple 403
- atomic exchange instruction 496
 - syntax summary 286
- AUX_DCCM 954
- AUX_ICCM 924
- AUX_IRQ_ACT 155
- AUX_IRQ_CTRL 153
- AUX_IRQ_HINT 157
- AUX_RTC_CTRL 1211
- AUX_RTC_HIGH 1214
- AUX_RTC_LOW 1213
- AUX_USER_SP 152
- auxiliary register instructions
 - summary 288
- auxiliary register map 886, 1273, 1362, 1444
- auxiliary register set 886, 953, 1273, 1362, 1444
 - 64-bit rtc 1211
 - actionpoint configuration 1130
 - active interrupts 155
 - barrel shifter configuration 195
 - BCR version 185, 199, 996
 - branch target address 179
 - BTA configuration 186
 - build configuration 84, 183, 981, 1270, 1337, 1439
 - core register set configuration 189
 - DCCM base address 954
 - DCCM RAM configuration 956, 957

exception cause 168
 exception fault address 177
 exception return address 165
 exception return branch target address 166, 1160
 exception return status 167
 execute indexed base address 164
 ICCM base address 924
 ICCM configuration 925
 identity 141
 individual interrupt pending 182
 instruction cache build 951
 interrupt build configuration 199
 interrupt cause 172
 interrupt context saving control 153
 interrupt enable 174
 interrupt P0 status link 151
 interrupt priority 158
 interrupt priority pending 156
 interrupt pulse cancel 181
 interrupt select 173
 interrupt status 176
 interrupt trigger 175
 interrupt vector base 154
 interrupt vector base address configuration 187
 ISA configuration 196
 JLI_BASE 162, 183
 LDI_BASE 163
 loop end 140
 loop start 139
 memory protection unit build configuration 878
 memory protection unit ECR 870
 memory protection unit enable 867
 MIN/MAX configuration 194
 MPU region descriptor base 873
 MPU region descriptor permissions 874
 multiply configuration 191
 normalize configuration 193
 peripheral memory region 1074
 processor timers 1201
 processor timers configuration 1215
 program counter 143
 rtc count high 1214
 rtc count low 1213
 saved user stack pointer 152
 smart control 986, 988, 990, 991, 994, 995, 1049, 1051, 1060, 1088, 1234, 1235, 1236, 1237
 software interrupt 157
 stack region build configuration 891
 status 145
 summary 97

swap configuration 192
 timer 0 control 1205
 timer 0 count 1204
 timer 0 limit 1207
 timer 1 control 1209
 timer 1 count 1208
 timer 1 limit 1210
 user mode extension enable 177
 auxiliary registers 288

B

B 410
 B_S 412
 barrel shift instructions
 syntax summary 276
 barrel shifter configuration register 195
 BARREL_BUILD 195
 basecase ALU instructions
 summary 273, 274, 275
 BBIT0 414
 BBIT1 416
 Bcc 418
 Bcc_S 420
 BCLR 422
 BCR version register 185, 199, 891, 996
 BCR_VER 185
 BH bit 1120
 BI 424
 BIC 426
 BIH 428
 bit clear instruction 422
 bit exclusive-or instruction 449
 bit mask instruction 433, 436
 bit set instruction 444
 bit test instruction 446
 bit toggle instruction 449
 bitwise and with invert instruction 426
 BLcc 430
 BLINK 131
 BMSK 433, 436
 branch and link instruction 430
 branch and link instructions
 syntax summary 292
 branch conditionally instruction 418
 16-bit format 420

branch indexed [424](#)
 branch indexed to half-word [428](#)
 branch instructions
 summary [291](#)
 syntax summary [291](#)
 branch on bit test instruction
 clear [414](#)
 syntax summary [292](#)
 branch on bit test set instruction
 set [416](#)
 branch on compare instruction [438](#)
 branch on compare instructions
 syntax summary [292](#)
 branch target address register [179](#)
 branch unconditionally instruction [410](#)
 BRcc [438](#)
 breakpoint instruction [441](#), [852](#)
 summary [296](#)
 trap 0 [783](#)
 BRK [441](#)
 BSET [444](#)
 BTA [179](#)
 BTA configuration register [186](#)
 BTA_LINK_BUILD [186](#)
 BTST [446](#)
 build configuration register
 actionpoint [1130](#), [1159](#)
 barrel shifter [195](#)
 BCR version [185](#)
 BTA [186](#)
 DCCM RAM [956](#)
 ICCM [925](#)
 instruction cache build [951](#)
 interrupt [199](#)
 interrupt vector base address [187](#)
 ISA [196](#)
 memory protection unit [878](#)
 MIN/MAX [194](#)
 multiply [191](#)
 normalize [193](#)
 processor timers [1215](#)
 stack region [891](#)
 build configuration registers [84](#), [183](#), [981](#), [1270](#),
[1337](#), [1439](#)
 BXOR [449](#)

C

canceling pulse triggered interrupts [215](#)
 CLRI [451](#)
 CMP [453](#)
 compare and branch instruction [438](#)
 compare instruction [453](#)
 reverse compare [715](#)
 condition codes [271](#)
 conditional branch [123](#)
 conditional branch and link instruction [430](#)
 conditional branch instruction [418](#)
 16-bit format [420](#)
 conditional execution [122](#)
 conditional jump and link instruction [591](#)
 conditional jump instruction [585](#)
 CONTROL0 [1205](#)
 core register map [125](#), [1329](#)
 core register set [125](#), [1329](#)
 extension registers [133](#), [1330](#)
 immediate data indicator [132](#)
 link registers [131](#)
 loop count [132](#)
 pointer registers [131](#)
 reduced configurations [128](#)
 reserved register r61 [132](#)
 summary [97](#)
 used in 16-bit instructions [127](#)
 COUNT0 [1204](#)

D

data formats [110](#)
 16-bit data [118](#)
 1-bit data [119](#)
 32-bit data [118](#)
 8-bit data [119](#)
 DCCM_BUILD [956](#), [957](#)
 DCCM_BUILD configuration register [956](#), [957](#)
 DEBUG [1118](#)
 debug register [1118](#)
 delayed load from memory instruction [599](#)
 DIV [457](#), [1524](#)
 DIVU [371](#), [375](#), [460](#), [463](#), [465](#), [468](#), [471](#), [477](#), [480](#),
[483](#), [662](#), [703](#), [706](#), [709](#), [712](#), [795](#), [798](#), [801](#), [804](#), [807](#),
[810](#), [817](#), [820](#), [823](#), [826](#), [829](#), [832](#), [834](#), [837](#), [844](#),
[1454](#), [1455](#), [1530](#), [1533](#), [1542](#), [1548](#), [1551](#), [1554](#),
[1557](#), [1566](#), [1569](#), [1572](#), [1574](#), [1577](#), [1583](#), [1655](#),

1731, 1734, 1737, 1740, 1743, 1746, 1778, 1785, 1787, 1789, 1791, 1793, 1800, 1802, 1804, 1807, 1854, 1864, 1910, 1912, 1914, 1916, 1918, 1920, 1922, 1924, 1926, 1928, 1934

DMP_PERIPHERAL 1074

double fault 249

DZ bit 148

E

ECR 168

EI_BASE 164

EI_S 148, 486, 490

emulation of extension condition codes 254

emulation of extension instructions 254

emulation of extension registers 254

encoding immediate data 132

ENTER_S 492

ERBTA 166

ERET 165

ERSTATUS 167

EX 496

exception cause register 168

exception fault address register 177

exception handling 250

exception return address register 165

exception return branch target address register 166, 1160

exception return status register 167

exceptions

cause codes 230

cause register 228, 230

data TLB miss 246

data TLB protection violation 246

delay slots 252

double fault 249

emulation of extension condition codes 254

emulation of extension instructions 254

emulation of extension registers 254

entry 249

exit 251

extension instruction 245

illegal instruction sequence 243

instruction error 241

instruction fetch protection violation 240, 246

instruction fetch TLB miss 246

machine check, double fault 240

machine check, fatal TLB error 240

machine check, instruction fetch memory error 246

machine check, overlapping TLB entries 240

overview 226

parameters 230

precision 226

priorities 230

privilege violation, action point hit on instruction fetch 241

privilege violation, actionpoint hit memory or register 248

privilege violation, disabled extension 245

privilege violation, kernel only access 245

privilege violation, misaligned data access 246

reset 239

types 230

vector name 228

vector number 228

vector offset 228

vectors 228

with interrupts 249

exchange instruction 496

syntax summary 286

exclusive-or instruction 846

EXTB 499

extend instruction

signed byte 756

signed half-word 758

unsigned byte 499

unsigned half-word 501

extension core registers 133, 1330

extension instructions

summary 297

extensions

auxiliary registers 106

condition codes 106

core register 106

instruction set 106

summary 106

extensions instruction

syntax summary 297

EXTH 501

F

features 95

FFS 545

FH bit 850, 1118

find first set 545

- find last set [550](#)
- FLAG [547](#)
- flag instruction [547](#)
- FLS [550](#)
- force halt [1118](#)
- FP [131](#)
- function epilog sequence [606](#)
- function prolog sequence [492](#)
- G**
- GP [131](#)
- H**
- H bit [150](#)
- halting the processor [850](#)
- host interface [849](#)
- I**
- I_CACHE_BUILD [951](#)
- I_CACHE_BUILD configuration register [951](#)
- ICAUSE [172](#)
- ICCM_BUILD [925](#)
- ICCM_BUILD configuration register [925](#)
- IDENTITY [141](#)
- identity register [141](#)
- IE bit [147](#)
- ILINK [131](#)
- immediate data indicator [132](#)
- instruction error [241](#)
- instruction format
 - summary [120](#)
- instruction set
 - 16 and 32 bit summary [98](#)
- instruction syntax convention [255](#)
- INT_VECTOR_BASE [154](#)
- interrupt
 - pending interrupts [212](#)
- interrupt (P0) status link registers [151](#)
- interrupt cause registers [172](#), [216](#)
- interrupt context saving control [153](#)
- interrupt priority pending register [156](#)
- interrupt unit configuration [209](#)
- interrupt unit programming [211](#)
- interrupt vector base address configuration register [187](#)
- interrupt vector base register [154](#)
- interrupts
 - canceling pulse triggered interrupts [215](#)
 - cause registers [216](#)
 - configuration [209](#)
 - programming [211](#)
 - returning [222](#)
 - sensitivity configuration [215](#)
 - software triggered [215](#)
 - timing [225](#)
- IRQ_ENABLE [174](#)
- IRQ_PENDING [182](#)
- IRQ_PRIORITY [158](#)
- IRQ_PRIORITY_PENDING [156](#)
- IRQ_PULSE_CANCEL [181](#)
- IRQ_SELECT [173](#)
- IRQ_STATUS [176](#)
- IRQ_TRIGGER [175](#)
- IS bit [851](#)
- ISA configuration register [196](#)
- ISA_CONFIG [196](#)
- J**
- J [582](#)
- Jcc [585](#)
- JL [588](#)
- JLcc [591](#)
- JLI_BASE [162](#), [183](#)
- JLI_S [594](#)
- jump and link conditionally instruction [591](#)
- jump and link indexed [594](#)
- jump and link unconditionally instruction [588](#)
- jump conditionally instruction [585](#)
- jump instructions
 - summary [294](#)
- jump unconditionally instruction [582](#)
- K**
- kernel flag instruction [596](#)
- kernel mode [205](#)
- KFLAG [596](#)
- L**
- L flag [149](#)
- LD [599](#)
- LD bit [1120](#)
- LDI [604](#)

LDI_BASE 163
LEAVE_S 606
limm 132
link register 217
link registers 131
list of Instructions 369, 1453
LLOCK 610, 612
load from auxiliary register instruction 621
 syntax summary 289
load from memory instruction 599
load pending 850, 1120
load register 288
load/store instructions
 summary 283
logical operations 297
logical shift left 625
logical shift left 16 623
logical shift right 634
logical shift right 16 632
logical shift right instruction 627
 multiple 629
Long immediate data
 ALU operations 297
 indicator 127
long immediate data
 16-bit instructions 352
 32-bit instructions 263, 370
loop count register 132
loop instruction
 syntax summary 294
loop set up instruction 455, 614
loops, zero-overhead 254
LP instruction 294
LP_COUNT 132, 294
LP_END 140, 294
LP_START 139, 294
LPcc 455, 614
LR 621
LSL16 623
LSL8 625
LSR 627
LSR multiple 629
LSR16 632
LSR8 634

M

MAX 648
maximum instruction 648
memory based instructions
 summary 283
memory error
 instruction fetch 246
memory prefetch instruction 695
MIN 651
MIN/MAX configuration register 194
minimum instruction 651
MINMAX_BUILD 194
MOV 654
move instruction 654, 1638, 1642
MPU_BUILD 878
MPU_BUILD configuration register 878
MPU_ECR 870
MPU_EN 867
MPY 656, 1644
MPYH 665, 1664
MPYHU 668, 1666
MPYU 671, 1668
MPYUW 673, 1670
MPYW 675, 1674
multiple arithmetic shift right instruction 403
multiply
 high result, signed 1664
 high result, unsigned 668, 1666
 signed 16x16 with 32-bit result 675, 1674
 unsigned 16x16 with 32-bit result 673
multiply configuration register 191
MULTIPLY_BUILD 191

N

NEG 677
negate instruction 677
no operation instruction 679
NOP_S 679
NORM 681
NORM_BUILD 193
normalize configuration register 193
normalize instruction 681
 word 683
NORMW 683
NOT 686

not instruction [686](#)

null instruction format [122](#)

O

operating modes [205](#)

summary [98](#)

switching [207](#)

operating systems [250](#)

OR [688](#)

or instruction [688](#)

P

PC [143](#)

PCL [133](#)

plotting results [1193](#)

pointer registers [131](#)

pop from stack instruction [691](#)

POP_S [691](#)

precise exceptions [226](#)

PREFETCH [695](#)

prefetch from memory instruction [695](#)

syntax summary [287](#)

privilege violations [206](#)

privileged instructions [206](#)

privileged registers [207](#)

privileges

introduction [205](#)

overview [206](#)

processor timers configuration register [1215](#)

program counter register [143](#)

programmer's model [97](#), [1422](#)

push onto stack instruction [701](#)

PUSH_S [701](#)

R

RA bit [1119](#)

RCMP [715](#)

register bank [1123](#)

REM [717](#), [1749](#)

REMU [720](#), [1752](#)

reserved register r61 [132](#)

reset [239](#)

return from interrupt/exception instruction [740](#)

summary [294](#)

returning from interrupts [222](#)

reverse compare instruction [715](#)

reverse subtract instruction [738](#)

RF_BUILD [189](#)

RLC [723](#)

ROL [725](#)

ROL8 [727](#)

ROR [729](#)

ROR multiple [731](#)

ROR8 [734](#)

rotate left [725](#)

rotate left through carry instruction [723](#)

rotate left8 [727](#)

rotate right 8 [734](#)

rotate right instruction [729](#)

multiple [731](#)

rotate right through carry instruction [736](#)

RRC [736](#)

RSUB [738](#)

RTIE [740](#)

RTOS [250](#)

S

SBC [744](#)

SCOND [746](#), [748](#)

self halt [1120](#)

SETcc [750](#)

SETEQ [751](#)

SETI [753](#)

SEXB [756](#)

SEXH [758](#)

SH bit [1120](#)

sign extend instruction

byte [756](#)

half-word [758](#)

signed 16x16 multiply

with 32-bit result [675](#), [1674](#)

signed multiply

high result [665](#), [1664](#)

single instruction step [851](#), [1119](#)

single operand extension instruction

syntax summary [298](#)

single step [851](#)

SLEEP [760](#), [840](#)

sleep instruction [760](#), [840](#), [1120](#)

summary [296](#)

SMART_CONTROL [1088](#), [1164](#), [1234](#), [1235](#), [1236](#),

1237

software interrupt instruction 783

software interrupt register 157

software triggered interrupt 215

SP 131

SR 764

ST 766

STACK_REGION_BUILD 891

STACK_REGION_BUILD configuration register 891

starting the processor 851

status register 145

status save register 217

STATUS32 145

store register 288

store to auxiliary register instruction 764

syntax summary 289

store to memory instruction 766

syntax summary 285

SUB 770

SUB1 773

SUB2 775

SUB3 777

subtract instruction 770

reverse 738

subtract with carry instruction 744

subtract with scaled source instruction

shift 1 773

shift 2 775

shift 3 777

SWAP 406, 408, 779

swap byte ordering 781

swap configuration register 192

swap instruction 406, 408, 779

SWAP_BUILD 192

SWAPE 781

SWI 783

switching operating modes 207

SYNC 786

synchronize instruction 786

summary 297

syntax summary

add instruction 273

atomic exchange instruction 286

auxiliary register instructions 288

barrel shift instructions 276

branch and link instructions 292

branch instructions 291

branch on bit test instruction 292

branch on compare instructions 292

exchange instruction 286

extension instructions 297

jump instructions 294

load from auxiliary register instruction 289

load/store instructions 283

loop instruction 294

memory based instructions 283

prefetch from memory instruction 287

single operand extension instruction 298

store to auxiliary register instruction 289

store to memory instruction 285

zero operand extension instruction 298

zero operand instructions 295

T

test instruction 791

timer 0 control register 1205

timer 0 count register 1204, 1213, 1214

timer 0 limit register 1207

timer 1 control register 1209

timer 1 count register 1208

timer 1 limit register 1210

Timer COUNT0 1204

TIMER_BUILD 1215

trap instruction 789

summary 296

trap 0 783

TRAP_S 789

TRAP0 783

TST 791

typographic convention 95

U

U flag 149

UB bit 1120

unconditional branch instruction

16-bit format 412

unconditional jump and link instruction 588

UNIMP_S 793

unimplemented instruction 793

unsigned 16x16 multiply

with 32-bit result 673

unsigned multiply
 high result [668](#), [1666](#)

user mode [205](#)

user mode extension enable register [177](#)

V

VECBASE_AC_BUILD [187](#)

X

XOR [846](#)

Z

zero delay loops [132](#), [294](#)

zero extend instruction

 byte [499](#)

 half-word [501](#)

zero operand extension instruction

 syntax summary [298](#)

zero operand instructions

 summary [295](#)

 syntax summary [295](#)

zero-overhead loop [254](#)

ZZ bit [1120](#)

