Synopsys® Processor IP

ARCv3 ISA - Public Programmer’s Reference Manual for ARC® HS6x Processors

Version 5820-001
August 2024

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

Synopsys, Inc.
www.synopsys.com

Contents

List of Figures 27

List of Examples 45

List of Tables 47

Preface 63

Quick Reference 67

Chapter 1
Build and Auxiliary Register List 69
1.1 Build Configuration Registers 75

Chapter 2 . . . . . .
Exceptions List 79

Part 1
ARCv3 Baseline ISA 85

Chapter 3
Introduction 87
3.1 Typographic Conventions 87
3.2 Key Features of the ARCv3 ISA 87
3.2.1 Programming Model 88
3.2.2 Core Register Set 88
3.2.3 Instruction Set Compatibility 88
3.2.4 Auxiliary Register Set 88
3.2.5 32-bit Instruction Formats 89
3.2.6 16-bit Instruction Formats 89
3.2.7 64-bit Data Models and Operating Modes 89
3.2.8 Operating Privileges 89
3.3 Configurability 90
3.4 Custom Extensions 94
3.4.1 Extension Core Registers 94
3.4.2 Extension Auxiliary Registers 95
3.4.3 Extension Instructions 95
3.4.4 Extension Condition Codes 95

Chapter 4

Data Organization and Addressing 97
4.1 The ARCv3 Address Space 97
4.2 Operations on 64-bit Addresses 98
4.3 Load and Store Instructions 98
4.3.1 64-bit Sign-extension for LDB.X and LDH.X 99
4.4 Data Formats 99
4.4.1 64-bit Data Processing Instructions 99
4.4.2 32-Bit Data Operations 99
4.4.3 Register Usage in 32-bit Instructions 99
4.4.4 Data Model for Size int 99
4.4.5 Vector Operands 100
4.4.6 32-Bit Operand Data Elements 100
4.4.7 Expansion of Long Immediate Literals 100
4.4.8 Expansion of Literals 101
4.4.9 Expansion of Literals in Vector Instructions 102
4.4.10 Expansion of Literals in Store Double (STD) 103
4.5 Data Layout in Memory 103
4.5.1 128-Bit Data 103
4.5.2 32-Bit Data 104
4.5.3 16-Bit Data 104
4.5.4 8-Bit Data 105
4.5.5 1-Bit Data 105
4.6 Instruction Layout in Memory 106
4.7 Addressing Modes 107
4.8 Null Instruction Format 108
4.9 Conditional Execution 108
4.10 Conditional Branch Instruction 109
4.11 Compare and Branch Instruction 109

Chapter 5
ARCv3 Memory Consistency Model 111
5.1 Terminology for Memory Ordering Rules 111
5.2 Memory Ordering 112

Chapter 6
Core Architectural State Registers 115
6.1 Core Register Set 115
6.1.1 Core Register Usage in the ABI 116
6.1.2 Function Prolog and Epilog Instructions 119
6.1.3 Multiple Register Banks 121
6.1.4 Illegal Core Register Usage 122
6.1.5 Pointer Registers, GP (r30), FP (r27), SP (r28) 122
6.1.6 Link Registers, ILINK (r29), BLINK (r31) 122
6.1.7 Immediate Data Indicator, LIMM (r62, r60) 123
6.1.8 Word-aligned Program Counter, PCL (r63) 123
6.2 Error Protection on Core Registers 123
6.3 Extension Core Registers 124
6.3.1 Illegal Extension Core Register Usage 124
6.4 Auxiliary Register Set 124

6.4.1 Baseline Auxiliary Registers . . . 124
6.4.2 Register Access Permissions . . . 125
6.4.3 Optional Instruction Set Auxiliary Registers . . . 126
6.4.4 User Extension Auxiliary Registers . . . 126
6.4.5 Optional Build Configuration Registers . . . 127
6.4.6 Auxiliary Registers Holding Address Values . . . 127
6.4.7 Core Identity Register, IDENTITY . . . 128
6.4.8 Program Counter, PC . . . 129
6.4.9 Status Register, STATUS32 . . . 130
6.4.10 Status Register Priority 0, STATUS32_P0 . . . 136
6.4.11 Saved User Stack Pointer, AUX_USER_SP . . . 137
6.4.12 Interrupt Context Saving Control Register, AUX_IRQ_CTRL . . . 138
6.4.13 Interrupt Vector Base Register, INT_VECTOR_BASE . . . 139
6.4.14 Active Interrupts Register, AUX_IRQ_ACT . . . 140
6.4.15 Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING . . . 141
6.4.16 Software Interrupt Trigger, AUX_IRQ_HINT . . . 142
6.4.17 Interrupt Priority Register, IRQ_PRIORITY . . . 143
6.4.18 Jump and Link Indexed Base Address, JLI_BASE . . . 144
6.4.19 Exception Return Address, ERET . . . 145
6.4.20 Exception Return Branch Target Address, ERBTA . . . 146
6.4.21 Exception Return Status, ERSTATUS . . . 147
6.4.22 Exception Cause Register, ECR . . . 148
6.4.23 Exception Fault Address, EFA . . . 149
6.4.24 Interrupt Cause Registers, ICAUSE . . . 150
6.4.25 Interrupt Select, IRQ_SELECT . . . 151
6.4.26 Interrupt Enable Register, IRQ_ENABLE . . . 152
6.4.27 Interrupt Trigger Register, IRQ_TRIGGER . . . 153
6.4.28 Interrupt Status Register, IRQ_STATUS . . . 154
6.4.29 User Mode Extension Enable Register, XPU . . . 155
6.4.30 Branch Target Address, BTA . . . 157
6.4.31 Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL . . . 158
6.4.32 Interrupt Pending Register, IRQ_PENDING . . . 159
6.4.33 User Extension Flags Register, XFLAGS . . . 160
6.5 Build Configuration Registers . . . 160
6.5.1 Build Configuration Registers Version, BCR_VER . . . 162
6.5.2 BTA Configuration Register, BTA_LINK_BUILD . . . 163
6.5.3 Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD . . . 164
6.5.4 Core Register File Configuration Register, RF_BUILD . . . 165
6.5.5 Multiplier Configuration Register, MULTIPLY_BUILD . . . 167
6.5.6 Swap Instruction Configuration Register, SWAP_BUILD . . . 168
6.5.7 Normalize Instruction Configuration Register, NORM_BUILD . . . 169
6.5.8 Min/Max Instruction Configuration Register, MINMAX_BUILD . . . 170
6.5.9 Barrel Shifter Configuration Register, BARREL_BUILD . . . 171
6.5.10 Instruction Set Configuration Register, ISA_CONFIG . . . 172
6.5.11 Interrupt Build Configuration Register, IRQ_BUILD . . . 174
6.5.12 ISA Profile Register, ISA_PROFILE . . . 176
6.5.13 Micro Architecture Configuration Register, MICRO_ARCH_BUILD . . . 178

Chapter 7

Interrupts and Exceptions . . . 179
7.1 Introduction . . . 179
7.2 Privileges and Operating Modes . . . 179
7.2.1 Kernel Mode . . . 179
7.2.2 User Mode . . . 179
7.2.3 Privilege Violations . . . 180
7.2.4 Switching Between Operating Modes and Privilege Levels . . . 181
7.3 Interrupts . . . 182
7.3.1 Interrupt Unit Features . . . 182
7.3.2 Interrupt Unit Configuration . . . 183
7.3.3 Interrupt Unit Programming . . . 183
7.3.4 Interrupt Handling . . . 188
7.4 Exceptions . . . 198
7.4.1 Exception Precision . . . 198
7.4.2 Exception Vectors and the Exception Cause Register . . . 199
7.4.3 Exception Types and Priorities . . . 201
7.4.4 Exception Detection . . . 217
7.4.5 Effect of Exceptions and Interrupts on Operating Mode . . . 217
7.4.6 Exception Entry . . . 217
7.4.7 Exception Exit . . . 219
7.4.8 Exceptions and Delay Slots . . . 220
7.4.9 Emulation of Extension Instructions . . . 222
7.4.10 Emulation of Extension Registers and Condition Codes . . . 222

Chapter 8
Instruction Set Summary . . . 223
8.1 Instruction Set Encoding . . . 223
8.1.1 Top-level Instruction Formats . . . 223
8.1.2 Instruction Set Profiles . . . 225
8.1.3 Assignment of Instruction Formats to Major Opcodes . . . 225
8.1.4 32-bit Instruction Formats . . . 228
8.1.5 16-bit Instruction Formats . . . 229
8.1.6 Encoding Notation . . . 230
8.1.7 Long Immediate Source Operands and Null Destination Operands . . . 232
8.1.8 Condition Code Tests . . . 233
8.1.9 Load or Store Instructions . . . 234
8.1.10 Encoding Branch and Jump Delay Slot Modes . . . 238
8.1.11 Semantics of Load / Store Address Write-back Modes . . . 239
8.1.12 Semantics of Load / Store Cache Bypass Mode . . . 240
8.1.13 Supported Load / Store Data Sizes . . . 240
8.1.14 Semantics of Load Data Extension Options . . . 240
8.1.15 Use of Reserved Encodings . . . 241
8.1.16 Use of Illegal Encodings . . . 241
8.2 Instruction Syntax Conventions . . . 241
8.3 Status flags and conditions . . . 243
8.3.1 Flag Setting . . . 243
8.3.2 Status Flags Notation . . . 243
8.4 Arithmetic and Logical Instructions . . . 244
8.4.1 Integer Adder Operations . . . 244

8.4.2 64-bit Integer Multiplication Operations . . . 246
8.4.3 Move Operations . . . 248
8.4.4 Bit-wise Logical Operations . . . 249
8.4.5 Bit Operations and Mask Operations . . . 251
8.4.6 Shift and Rotate Operations . . . 251
8.4.7 Selection Operations . . . 254
8.4.8 Byte-swapping Operations . . . 254
8.4.9 Bit-scanning Operations . . . 255
8.4.10 Relational Comparison Operations . . . 255
8.4.11 Integer Multiply, Multiply-accumulate, and Divide Operations . . . 256
8.4.12 Dual and Quad Integer Multiply / Accumulate Operations . . . 257
8.4.13 Integer Vector Operations . . . 259
8.5 Memory Access Instructions . . . 260
8.5.1 Load Instructions . . . 261
8.5.2 Store Instructions . . . 262
8.5.3 Stack Pointer Operations . . . 263
8.5.4 Atomic Memory Operations . . . 263
8.5.5 Prefetch Instructions . . . 266
8.6 Auxiliary Register Operations . . . 267
8.6.1 Load from Auxiliary Register . . . 268
8.6.2 Store to Auxiliary Register . . . 268
8.6.3 Auxiliary Register Exchange . . . 268
8.7 Control Flow Instructions . . . 268
8.7.1 Branch Instructions . . . 270
8.7.2 Return from Interrupt or Exception Instruction . . . 273
8.8 Vector Pack Instructions . . . 273
8.9 Special Instructions . . . 274
8.9.1 Breakpoint Instruction . . . 275
8.9.2 Sleep Instruction . . . 275
8.9.3 Software Interrupt Instruction . . . 275
8.9.4 SETI . . . 276
8.9.5 CLRI . . . 276
8.9.6 Trap Instruction . . . 276
8.9.7 Synchronization Instructions . . . 276
8.10 APEX Extension Instructions . . . 276
8.10.1 Syntax for Generic Extension Instructions . . . 276
8.10.2 Syntax for Single Operand Extension Instructions . . . 277
8.10.3 Syntax for Zero Operand Extension Instructions . . . 277
8.11 Serializing Instructions and Events . . . 278
8.11.1 Instruction Barrier . . . 278
8.11.2 Serializing Instructions . . . 278
8.11.3 Serializing Events . . . 279
8.11.4 Serializing Instructions in ARCv2 and ARCv3 . . . 279
8.11.5 Serializing Events in ARCv2 and ARCv3 . . . 279

Chapter 9
32-bit Instruction Formats Reference . . . 281
9.1 Branch Format [F32_BR0] . . . 282
9.1.1 Branch-Conditional Sub-format [F32_BR0, COND] . . . 282

9.1.2 Branch Unconditional Far Sub-format [F32_BR0, UCOND_FAR] . . . 282
9.2 BRcc, BBITn and BL Format [F32_BR1] . . . 283
9.2.1 BRcc / BBITn Register-register Format [F32_BR1, BCC, REG_REG] . . . 283
9.2.2 Register-immediate Format [F32_BR1, BCC, REG_U6] . . . 284
9.2.3 Branch-and-link Format [F32_BR1, BL] . . . 284
9.3 Load Register with Offset Format [F32_LD_OFFSET] . . . 286
9.4 Store Register with Offset Format [F32_ST_OFFSET] . . . 287
9.5 General Operations Format [F32_GEN4] . . . 288
9.5.1 Operator Format Indicators . . . 288
9.5.2 Operand Format Indicators . . . 288
9.5.3 Syntax and Encoding of Instructions in General Operations Formats . . . 289
9.5.4 ARC64 Extension Instruction Format [F32_GEN_OP64] . . . 290
9.5.5 Single Operand Instructions, F32_GEN4 . . . 293
9.5.6 Zero Operand Instructions, F32_GEN4 . . . 293
9.5.7 Dual-operand Instructions, F32_GEN_OP64 . . . 294
9.5.8 Single-operand Instructions, F32_GEN_OP64 . . . 295
9.5.9 Zero-operand Instructions, F32_GEN_OP64 . . . 295
9.5.10 Move and Compare Instructions, 0x04, [0x0A – 0x0D] and 0x04, [0x11] . . . 296
9.5.11 Jump and Jump-and-Link Conditionally, 0x04, [0x20 – 0x23] . . . 296
9.5.12 Load Register-Register, 0x04, [0x30 – 0x37] . . . 296
9.6 F32_EXT5 . . . 297
9.6.1 Dual-operand Instructions, F32_EXT5 . . . 297
9.6.2 Single-operand Instructions, F32_EXT5 . . . 298
9.6.3 Zero-operand Instructions, F32_EXT5 . . . 299
9.7 APEX Extension Instruction Format [F32_APEX] . . . 299
9.7.1 Extension ALU Operation, Register with Unsigned 6-bit Immediate . . . 299
9.7.2 Extension ALU Operation, Register with Signed 12-bit Immediate . . . 300
9.7.3 Extension ALU Operation, Conditional Register . . . 300
9.7.4 Extension ALU Operation, Conditional Reg with Unsigned 6-bit Immediate . . . 300
9.7.5 FastMath Extension Pack Instructions . . . 301
9.8 ARC64 Extension Instruction Format [F32_GEN_OP64] . . . 302

Chapter 10
16-bit Instruction Formats Reference . . . 303
10.1 Compact Move [F16_COMPACT0] . . . 305
10.2 Compact MOVL [F16_MOVL] . . . 306
10.3 Compact Load/Add/Sub [F16_LD_ADD_SUB] . . . 307
10.4 Compact Load/Store [F16_LD_ST_JLI] . . . 308
10.5 Load /Add Register-Register [F16_LD_ADD_RR] . . . 309
10.6 Dual Register Operations [F16_OP_HREG] . . . 310
10.6.1 Assign a 64-bit Literal to a GPR . . . 311
10.6.2 Assign PCL + 64-bit literal to a GPR . . . 311
10.7 General Register Format Instructions [F16_GEN_OP] . . . 312
10.7.1 DOP-format General Operations . . . 312
10.7.2 SOP-format 16-bit General Operations . . . 313
10.7.3 ZOP-format 16-bit General Operations . . . 314
10.8 16-bit Load and Store Formats with Offset . . . 316
10.9 Shift/Subtract/Bit Immediate [F16_SH_SUB_BIT] . . . 317
10.10 Stack-based Operations [F16_SP_OP64] . . . 318

10.11 Load/Add GP-Relative [F16_GP_OP64] . . . 319
10.12 Load PCL-Relative [F16_PCL_LD] . . . 320
10.13 Move Immediate [F16_MV_IMM] . . . 321
10.14 Branch on Compare Register with Zero [F16_BCC_REG] . . . 322
10.15 Branch Conditionally [F16_BCC] . . . 323
10.15.1 Branch Conditionally with cc Field, 0x1E, [0x03, 0x00 – 0x07] . . . 323
10.16 Branch and Link Unconditionally [F16_BL] . . . 325
10.17 Compact Long-jump and Long-call Instructions . . . 326

Chapter 11
Instruction Set Details . . . 329
11.1 Instruction List . . . 329
11.2 ARC Instructions . . . 340

Chapter 12
The Host . . . 863
12.1 The Host Interface . . . 863
12.2 Core and Auxiliary Registers . . . 864
12.3 Memory . . . 864
12.4 Halting . . . 864
12.5 Starting . . . 865
12.6 Single Instruction Stepping . . . 865
12.6.1 SLEEP Instruction in Single Instruction Step Mode . . . 866
12.6.2 BRK Instruction in Single Instruction Step Mode . . . 866
12.7 Software Breakpoints . . . 866

Part 2
Memory and System Components . . . 867

Chapter 13
Memory Protection Unit . . . 869
13.1 Memory Protection Unit . . . 869
13.1.1 Key Features . . . 870
13.1.2 Configuration Options . . . 870
13.1.3 Enabling the MPU . . . 870
13.1.4 Data Organization and Addressing . . . 870
13.1.5 Modes of Operation . . . 873
13.1.6 Memory Protection Unit Registers . . . 878
13.1.7 MPU Enable Register, MPU_EN . . . 881
13.1.8 MPU Exception Cause Register, MPU_ECR . . . 884
13.1.9 MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_RDB31 . . . 887
13.1.10 MPU Region Descriptor Permissions Registers, MPU_RDP0 to MPU_RDP31 . . . 888
13.1.11 MPU Attribute Register, MPU_MEM_ATTR . . . 891
13.1.12 Memory Protection Build Configuration Register, MPU_BUILD . . . 893
13.1.13 MPU Exceptions . . . 893

Chapter 14
Protection Schemes . . . 895
14.1 Stack Checking . . . 895
14.1.1 Build Configuration Register . . . 895

14.1.2 Enabling Stack Checking . . . 895
14.1.3 Specifying Stack Regions . . . 896
14.1.4 Stack-Checking Operation . . . 897
14.1.5 Exception Information . . . 900
14.1.6 Stack Protection Out-of-Bound Limitations . . . 900
14.1.7 Stack Region Auxiliary Register Set . . . 900
14.1.8 Stack Checking Exceptions . . . 905
14.2 Coexisting Protection Schemes . . . 906

Chapter 15
Memory Management Unit . . . 907
15.1 MMU Introduction . . . 907
15.2 High-Level Architecture . . . 907
15.2.1 MMUv48 . . . 907
15.2.2 MMUv52 . . . 911
15.2.3 Table, Block and Page Attributes . . . 914
15.3 Programming Model . . . 919
15.3.1 Address Spaces . . . 919
15.4 Translation Lookaside Buffer (TLB) . . . 920
15.4.1 TLB Lookup Flowchart . . . 920
15.5 Hardware Page Table Walk . . . 922
15.6 MMU Registers . . . 923
15.6.1 MMU Root Translation Pointer0 Auxiliary Register, MMU_RTP0 . . . 924
15.6.2 MMU Root Translation Pointer1 Auxiliary Register, MMU_RTP1 . . . 926
15.6.3 MMU TLB Index Register, MMU_TLB_IDX . . . 928
15.6.4 MMU TLB Command Register, MMU_TLB_CMD . . . 930
15.6.5 MMU TLB Data 0 Register, MMU_TLB_DATA0 . . . 932
15.6.6 MMU TLB Data 1 Register, MMU_TLB_DATA1 . . . 933
15.6.7 MMU Control Register, MMU_CTRL . . . 934
15.6.8 MMU Translation Table Base Control Register, MMU_TTBC . . . 935
15.6.9 MMU Attribute Register, MMU_MEM_ATTR . . . 937
15.6.10 MMU Build Configuration Register, MMU_BUILD . . . 939
15.7 Software Sequence For Updating Page Tables . . . 940

Chapter 16
ICCM . . . 943
16.1 ICCM Auxiliary Registers . . . 943
16.1.1 ICCM base address, AUX_ICCM . . . 944
16.1.2 ICCM Configuration Register, ICCM_BUILD . . . 945
16.2 Exceptions . . . 946

Chapter 17
Instruction Cache . . . 947
17.1 Instruction Cache Auxiliary Registers . . . 947
17.2 Auxiliary Registers . . . 947
17.2.1 Invalidate Instruction Cache, IC_IVIC . . . 949
17.2.2 Instruction Cache Control Register, IC_CTRL . . . 950
17.2.3 Lock Instruction Cache Line, IC_LIL . . . 951
17.2.4 Invalidate Instruction-Cache Start Region, IC_IVIR . . . 953
17.2.5 Invalidate Instruction-Cache End Region, IC_ENDR . . . 955

17.2.6 Invalidate Instruction-Cache Line, IC_IVIL 956
17.2.7 Instruction Cache External-Access Address, IC_RAM_ADDR 957
17.2.8 Instruction-Cache Tag Access, IC_TAG 960
17.2.9 Instruction Cache Data Access, IC_DATA 964
17.2.10 Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG 967
17.2.11 Instruction Cache Way-Lock Register, IC_WAYLOCK 968
17.2.12 Instruction Cache Configuration Register, I_CACHE_BUILD 969
17.3 Exceptions 970

Chapter 18
DCCM 971
18.1 DCCM Auxiliary Registers 971
18.1.1 DCCM Base Address, AUX_DCCM 972
18.1.2 DCCM Configuration Register, DCCM_BUILD 973
18.1.3 Exceptions 974

Chapter 19
Data Cache 975
19.1 Data Cache Auxiliary Registers 975
19.1.1 Invalidate Data Cache, DC_IVDC 977
19.1.2 Data Cache Control Register, DC_CTRL 979
19.1.3 Lock Data Cache Line, DC_LDL 981
19.1.4 Invalidate Data Line, DC_IVDL 982
19.1.5 Flush Data Cache, DC_FLSH 983
19.1.6 Flush Data Line, DC_FLDL 984
19.1.7 Data-Cache Region Start Address, DC_STARTR 985
19.1.8 Data-Cache Region End Address, DC_ENDR 986
19.1.9 Data Cache External Access Address, DC_RAM_ADDR 987
19.1.10 Data Cache Tag Access, DC_TAG 990
19.1.11 Data Cache Data Access, DC_DATA 992
19.2 Build Configuration Registers 992
19.2.1 Data Cache Configuration Register, D_CACHE_BUILD 993
19.2.2 Exceptions 994
19.3 Cache Maintenance 994
19.3.1 Cache-based Purge and Flush Operations 995
19.3.2 Region-based Flush 995
19.3.3 Region-Based Purge 995

Chapter 20
Error Protection 997
20.1 Auxiliary Registers 997
20.1.1 ECC SBE Counter Register, ECC_SBE_COUNT 998
20.1.2 Error Protection Hardware Control Register, ERP_CTRL 1001
20.1.3 Error Protection Configuration Register, ERP_BUILD 1003
20.2 Exceptions 1005

Chapter 21
Peripheral Bus 1007
21.1 DMP Auxiliary Registers 1007
21.1.1 Private Peripheral Aperture Base Address, PER_BASE 1008

21.1.2 Private Peripheral Aperture Size, PER_SIZE 1009
21.1.3 Peripheral Region Build Configuration Register, DMP_PER_BUILD 1010
21.2 Programming Notes 1010

Chapter 22
Write Buffer 1011
22.1 Introduction 1011
22.2 Configuring the Write Buffer 1011
22.3 Write Buffer Auxiliary Registers 1011
22.3.1 Write Buffer Control Register, WB_CTRL 1012
22.3.2 Write Buffer Status Register, WB_STATUS 1013
22.3.3 Write Buffer BCR Register, WB_BUILD 1014

Chapter 23
Branch Prediction Unit 1015
23.1 Branch Predication Unit Introduction 1015
23.2 Branch Predication Unit Register Interface 1015
23.2.1 Flush and Initialize Branch Predictor Register, BPU_FLUSH 1016
23.2.2 BPU Control Register, BPU_CTRL 1017
23.2.3 Branch Cache RAM Address Register, BPU_RAM_ADDR 1018
23.2.4 Branch Cache Information Register, BC_RAM_INFO 1019
23.2.5 Branch Cache Branch Target Address Register, BC_RAM_BTA 1021
23.2.6 Prediction Table DATA Register, PT_RAM_DATA 1022
23.2.7 Branch Prediction Unit Configuration Register, BPU_BUILD 1023
23.3 Debugging BPU 1024
23.3.1 Best Practices to Follow When Debugging BPU 1024
23.3.2 How to Dump Branch Cache and Prediction Table Data 1024
23.3.3 Usage Examples 1024

Chapter 24
External Host Debugging 1027
24.1 External Host Debugging Introduction 1027
24.2 Host Debug Register Interface 1027
24.3 Debug Register, DEBUG 1028

Chapter 25
Debug Features 1033
25.1 Debug Introduction 1033
25.2 Actionpoints 1033
25.2.1 Actionpoint Auxiliary Registers 1033
25.2.2 Actionpoint Extensions to Debug Register, DEBUG, 0x05 1037
25.2.3 Actionpoints Configuration Register, AP_BUILD 1038
25.2.4 Actionpoint Match Value, AP_AMV0 1038
25.2.5 Actionpoint Match Mask, AP_AMM0 1039
25.2.6 Actionpoint Control, AP_AC0 1040
25.2.7 Actionpoint Match Value, AP_AMV1 1044
25.2.8 Actionpoint Match Mask, AP_AMM1 1044
25.2.9 Actionpoint Control, AP_AC1 1045
25.2.10 Actionpoint Match Value, AP_AMV2 1045
25.2.11 Actionpoint Match Mask, AP_AMM2 1045

25.2.12 Actionpoint Control, AP_AC2 . . . 1046
25.2.13 Actionpoint Match Value, AP_AMV3 . . . 1046
25.2.14 Actionpoint Match Mask, AP_AMM3 . . . 1046
25.2.15 Actionpoint Control, AP_AC3 . . . 1047
25.2.16 Actionpoint Match Value, AP_AMV4 . . . 1047
25.2.17 Actionpoint Match Mask, AP_AMM4 . . . 1047
25.2.18 Actionpoint Control, AP_AC4 . . . 1047
25.2.19 Actionpoint Match Value, AP_AMV5 . . . 1048
25.2.20 Actionpoint Match Mask, AP_AMM5 . . . 1048
25.2.21 Actionpoint Control, AP_AC5 . . . 1048
25.2.22 Actionpoint Match Value, AP_AMV6 . . . 1049
25.2.23 Actionpoint Match Mask, AP_AMM6 . . . 1049
25.2.24 Actionpoint Control, AP_AC6 . . . 1049
25.2.25 Actionpoint Match Value, AP_AMV7 . . . 1050
25.2.26 Actionpoint Match Mask, AP_AMM7 . . . 1050
25.2.27 Actionpoint Control, AP_AC7 . . . 1050
25.2.28 Watchpoint Program Counter, AP_WP_PC . . . 1051
25.3 SmaRT . . . 1051
25.3.1 Features . . . 1051
25.3.2 Overview of SmaRT Operation . . . 1052
25.3.3 Auxiliary Registers . . . 1053
25.3.4 SMART_BUILD Configuration Register, SMART_BUILD . . . 1053
25.3.5 SmaRT Control Register, SMART_CONTROL . . . 1054
25.3.6 SmaRT Data Register, SMART_DATA . . . 1055
25.3.7 SRC_ADDR Value, Index 0 . . . 1055
25.3.8 DEST_ADDR Value, Index 1 . . . 1056
25.3.9 FLAGS Value, Index 2 . . . 1056
25.4 Soft Reset . . . 1057
25.4.1 Soft Reset Capability for Debug Host . . . 1057
25.4.2 Interrupt and Register Bank Debug Register, DEBUGI . . . 1058
25.4.3 Core State After Soft Reset . . . 1059
25.4.4 Soft Reset Auxiliary Registers . . . 1059
25.4.5 CLN Soft Reset Cores, CLN_SOFTRESET_CORES . . . 1060
25.4.6 CLN Soft Reset Device Ports, CLN_SOFTRESET_DEV . . . 1062
25.4.7 Soft Reset PC, SOFT_RESET_PC . . . 1063
25.4.8 Soft Reset STATUS32, SOFT_RESET_STATUS32 . . . 1064
25.4.9 Soft Reset State, SOFT_RESET_STATE . . . 1065
25.4.10 Soft Reset Saved ECR, SOFT_RESET_ECR . . . 1067
25.4.11 Soft Reset Save Active Interrupt Register, SOFT_RESET_IRQ_ACT . . . 1068
25.5 Debug Interface . . . 1068
25.5.1 ARC Debug-Access Registers . . . 1068
25.5.2 Debug Status Register, DB_STATUS . . . 1069
25.5.3 Debug Command Register, DB_CMD . . . 1070
25.5.4 Debug Address Register, DB_ADDR . . . 1070
25.5.5 Debug Address Register, DB_ADDR_HI . . . 1071
25.5.6 Debug Data Register, DB_DATA . . . 1071
25.5.7 Debug Data Register, DB_DATA_HI . . . 1071
25.5.8 Debug Reset Register, DB_RESET . . . 1072

Chapter 26
Performance Counters . . . 1075
26.1 Performance Counters Introduction . . . 1075
26.2 Performance Counters Registers . . . 1076
26.2.1 Countable Conditions Index Register, CC_INDEX . . . 1078
26.2.2 Countable Conditions Name0 Register, CC_NAME0 . . . 1079
26.2.3 Countable Conditions Name1 Register, CC_NAME1 . . . 1080
26.2.4 Count-Value Registers, PCT_COUNTL . . . 1081
26.2.5 Snapshot-Value Registers, PCT_SNAPL . . . 1082
26.2.6 Configuration Register, PCT_CONFIG . . . 1083
26.2.7 Control Register, PCT_CONTROL . . . 1084
26.2.8 Index-Select Register, PCT_INDEX . . . 1086
26.2.9 Address-Range Registers, PCT_RANGEL . . . 1087
26.2.10 Address-Range Registers, PCT_RANGEH . . . 1088
26.2.11 User-Flag Register, PCT_UFLAGS . . . 1089
26.2.12 Interrupt Trigger Value, PCT_INT_CNTL . . . 1091
26.2.13 Interrupt Control Value, PCT_INT_CTRL . . . 1092
26.2.14 Interrupt Active, PCT_INT_ACT . . . 1093
26.2.15 Performance Counter Build-Configuration Register, PCT_BUILD . . . 1094
26.2.16 Countable Conditions Build Configuration Register, CC_BUILD . . . 1095

Chapter 27
Processor Timers . . . 1097
27.1 Timers . . . 1097
27.1.1 Programming . . . 1097
27.2 Auxiliary Timer Registers . . . 1098
27.2.1 Timer 0 Count Register, COUNT0 . . . 1100
27.2.2 Timer 0 Control Register, CONTROL0 . . . 1101
27.2.3 Timer 0 Limit Register, LIMIT0 . . . 1103
27.2.4 Timer 1 Count Register, COUNT1 . . . 1104
27.2.5 Timer 1 Control Register, CONTROL1 . . . 1105
27.2.6 Timer 1 Limit Register, LIMIT1 . . . 1106
27.2.7 RTC Control Register, AUX_RTC_CTRL . . . 1107
27.2.8 RTC Count Register, AUX_RTC_LOW . . . 1108
27.2.9 Timers Configuration Register, TIMER_BUILD . . . 1109

Chapter 28
ARC Trace . . . 1111
28.1 ARC Trace Introduction . . . 1111
28.2 ARC Trace Auxiliary Registers . . . 1111
28.2.1 ARC Trace Address Register, RTT_ADDRESS . . . 1112
28.2.2 ARC Trace DATA Register, RTT_DATA . . . 1113
28.2.3 ARC Trace CMD Register, RTT_COMMAND . . . 1114
28.2.4 ARC Trace Build Configuration Register, RTT_BUILD . . . 1115
28.3 ARC Trace Address Map . . . 1116
28.4 ARC Trace Producer Registers . . . 1116
28.4.1 Register Description . . . 1118
28.4.2 ARC Trace I/F Build Configuration Register, ARCT_BUILD . . . 1119
28.4.3 Trace Producer Select Register, ARCT_PRSEL . . . 1121

28.4.4 Trace Nexus Flush Command Register, ARCT_FLUSH_COMMAND 1122
28.4.5 Trace Timestamp Enable Register, ARCT_TSTAMP_ENABLE 1123
28.4.6 Trace Debugger Message Enable Register, ARCT_DEBUGGER_MESSAGE_ENABLE 1124
28.4.7 Producer System Timestamp Enable Register, PR_SYTM_ENABLE 1125
28.4.8 Producer CoreSight Timestamp Register, CSTEN_REG_ADDR 1126
28.4.9 Producer Cross Trigger Enable Register, CTIEN_REG_ADDR 1127
28.4.10 Producer Trace Attribute Register, ARCT_PRATTR 1128
28.4.11 Trace Producer Source Enable Register, PR_SRC_EN 1130
28.4.12 EVTI Control Register, PR_EVTI_REG 1131
28.4.13 ARC Trace Producer Build Configuration Register, RTT_PRDCR_BCR 1132
28.4.14 Address Filter Registers 1132
28.4.15 Trace Producer Type Register, ARCT_PRTYPE 1136
28.4.16 Trace Filter Type Register, ARCT_FLT_TYPE 1137
28.4.17 Trace Filter Control Register, ARCT_FLT_CTRL 1138
28.4.18 Trace Producer Watchpoint Status Register, PR_WP_STATUS 1140
28.4.19 Trace Producer Watchpoint Enable Register, PR_WP_ENABLE 1142
28.5 ARC Trace Transport Status and Control Registers 1143
28.5.1 Trace Nexus Flush Command Register, ARCT_FLUSH_COMMAND 1144
28.5.2 Producer CoreSight ID Register, ATID_REG_ADDR 1145
28.5.3 Producer ATB SYNC Frame Insertion Register, SYNCRFR_REG_ADDR 1146
28.5.4 Trace On-Chip Memory Base Address Register, ARCT_OCM_BASE 1147
28.5.5 Trace On-Chip Memory Size Register, ARCT_OCM_SIZE 1148
28.5.6 Trace On-Chip Memory Write Pointer Register, ARCT_OCM_WPTR 1149
28.5.7 Trace Transport Status Register, ARCT_TR_STATUS 1150
28.5.8 Trace Nexus Offload Control Register, ARCT_OFFLOAD_CTRL 1152
28.5.9 Pattern Generation Register, PTRN_GEN 1153
28.5.10 Nexus Clock Control Register, NEXUS_CLK_DIV 1154

Chapter 29
Cluster Network 1155
29.1 Hardware/Software Interface 1155
29.1.1 Cluster Build Configuration Register, CLUSTER_BUILD 1156
29.1.2 Register Overview 1156
29.2 Modifying the Cluster Network Configuration 1178
29.2.1 Shared Cache and Memory Active Configuration Register 1, CLN_SCM_ACR_1 1179
29.3 Programming the Shared Memory Aperture 1179
29.3.1 Shared Memory Aperture Lowest Address Register, CLN_SHMEM_ADDR 1180
29.3.2 Shared Memory Aperture Size Register, CLN_SHMEM_SIZE 1181
29.4 Flushing and Invalidating the Shared Cache 1181
29.4.1 Lowest Address of a Range for CLN_CACHE_ADDR_LO0 [31:6] Register 1182
29.4.2 Lowest Address of a Range for CLN_CACHE_ADDR_LO1 [51:32] Register 1183
29.4.3 Highest Address of a Range for CLN_CACHE_ADDR_HI0 [31:0] Register 1184
29.4.4 Highest Address of a Range for CLN_CACHE_ADDR_HI1 [51:32] Register 1185
29.4.5 Cache Operation Command Register, CLN_CACHE_CMD 1186
29.4.6 Cache Status and Operation Result Register, CLN_CACHE_STATUS 1189
29.4.7 Cache Writeback Error Status Register, CLN_CACHE_ERR 1190
29.4.8 Faulting Address Bits [31:0] Register, CLN_CACHE_ERR_ADDR0 1191
29.4.9 Faulting Address Bits [51:32] Register, CLN_CACHE_ERR_ADDR1 1192
29.5 Programming Master Port Address Apertures 1192

29.5.1 NoC Master #i Aperture #j Lowest Address Register, CLN_MST_NOC_i_j_ADDR 1193
29.5.2 Size NoC Master #i Aperture #j Size Register, CLN_MST_NOC_i_j_SIZE 1194
29.5.3 CCM Master #i Aperture #k Lowest Address Register, CLN_MST_CCM_i_k_ADDR 1195
29.5.4 CCM Master #i Aperture #k Size Register, CLN_MST_CCM_i_k_SIZE 1195
29.5.5 PERj Peripheral Aperture Lowest Address Register, CLN_PERj_BASE 1197
29.5.6 PERj Peripheral Aperture Size Register, CLN_PERj_SIZE 1198
29.5.7 Global Clock Disable Register, GLOBAL_CLK_GATE_DIS 1199
29.5.8 ARC Cores Power Status Register, CLN_PWR_STATUS_0 1202
29.5.9 Initiator Port Power Status Register, CLN_PWR_STATUS_1 1203
29.5.10 Device Target Port Power Status Register, CLN_PWR_STATUS_2 1204
29.5.11 ARC Trace Power Status Register, CLN_RTT_PDM_STATUS 1205
29.5.12 Programming ARC Trace Transport Parameters 1205
29.5.13 ARC Trace Transport Register, CLN_SLV_i_ARCT 1206
29.5.14 Programming QoS Parameters 1206
29.5.15 QOS Control Common to all Resource Domains Register, CLN_QOS_CTL 1207
29.6 Error Conditions and Job Completion 1209
29.6.1 Posted Write Error Status Register, CLN_WR_ERR 1210
29.6.2 Posted Write Error Address Bits [31:0] Register, CLN_WR_ERR_ADDR0 1211
29.6.3 Posted Write Error Address Bits [51:32] Register, CLN_WR_ERR_ADDR1 1212
29.6.4 'attn' Interrupt Signal Status Register, CLN_ATTN 1213
29.7 Error Protection Registers 1213
29.7.1 Data Bank ECC Control Register, CLN_DBANK_ECC_CTRL 1214
29.7.2 Tag Bank ECC Control Register, CLN_TBNK_ECC_CTRL 1216
29.7.3 SCU Shadow Tag Memory ECC Control Register, CLN_STAG_ECC_CTRL 1218
29.7.4 DMA Descriptor Memory ECC Control Register, CLN_CDMA_ECC_CTRL 1220
29.8 Cluster Auxiliary and Build Configuration Registers 1221
29.8.1 CLUSTER ID Register, CLUSTER_ID 1222
29.9 Cluster Performance Counter Registers 1222
29.9.1 Cluster Performance Counter Build Configuration Register, CPCT_BUILD 1224
29.9.2 Index-Select Register: CPCT_INDEX 1227
29.9.3 Control Register: CPCT_CONTROL 1228
29.9.4 Interrupt Control Register, CPCT_INT_CTRL 1229
29.9.5 Interrupt Active Register, CPCT_INT_ACT 1230
29.9.6 Configuration Register: CPCT_<N>CONFIG 1231
29.9.7 Count-Value Registers: CPCT_COUNTL 1233
29.9.8 Count-Value Registers: CPCT_COUNTH 1234
29.9.9 Snapshot-Value Registers, CPCT_<N>_SNAPL 1235
29.9.10 Interrupt Trigger Value Registers, CPCT_INT_CNTL 1236
29.9.11 Interrupt Trigger Value Registers, CPCT_INT_CNTH 1237
29.9.12 Programming Sequence 1237
29.9.13 Programming in a Multi-core Processor 1238

Chapter 30
Hardware Prefetching 1241
30.1 Programming Model / Auxiliary Registers 1241
30.1.1 Hardware Prefetcher Control Auxiliary Register, HW_PF_CTRL 1242
30.2 Software Prefetching 1243
30.3 Speculative Accesses 1243
30.4 Fences and Memory Barriers 1243

Chapter 31
Power Management Features 1245
31.1 Introduction to Power Management Features 1245
31.2 PDM and DVFS Register Interface 1245
31.2.1 Core Power Status Register, PDM_PSTAT 1246
31.2.2 ARC Trace Power Status Register, RTT_PDM_PSTAT 1248
31.2.3 Power Down Register, PDM_PMODE 1249
31.3 Execution Rate Interface 1250
31.3.1 Execution Rate Control Register, EXEC_CTRL 1251
31.4 Build Configuration Registers 1251
31.4.1 Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD . 1253

Chapter 32
ARConnect 1255
32.1 ARConnect Introduction 1255
32.2 ARConnect Register Interface 1255
32.2.1 ARConnect Command Register, CONNECT_CMD 1257
32.2.2 ARConnect Write Data Register, CONNECT_WDATA 1258
32.2.3 ARConnect Read Data Register, CONNECT_READBACK 1259
32.2.4 ARConnect Read 64-Bit Data Register, CONNECT_READBACK _64 1260
32.2.5 ARConnect Build Configuration Register, CONNECT_SYSTEM_BUILD 1261
32.2.6 Inter-core Semaphore Unit BCR, CONNECT_SEMA_BUILD 1263
32.2.7 Inter-core Message Unit BCR, CONNECT_MESSAGE_BUILD 1264
32.2.8 Interrupt Distribution Unit BCR, CONNECT_IDU_BUILD 1265
32.2.9 ARConnect Global Free Running Counter BCR, CONNECT_GFRC_BUILD 1266
32.2.10 Inter-Core Interrupt Unit BCR, CONNECT_ICI_BUILD 1267
32.2.11 Inter-Core Debug Unit BCR, CONNECT_ICD_BUILD 1268
32.3 Programming Restrictions 1268
32.3.1 Atomic Operation 1268
32.3.2 Preventing Conflicts 1268

Chapter 33
Cluster DMA Controller 1271
33.1 High-Level Architecture 1271
33.2 Example Use Cases 1275
33.2.1 Polling Mode 1276
33.2.2 Interrupt Mode 1277
33.2.3 Event Mode 1278
33.2.4 Low level API 1279
33.2.5 Context Switching 1281
33.3 Configuration Options 1282
33.4 Programming the Cluster DMA using Auxiliary Registers 1282
33.4.1 DMA Build Configuration Register, DMA_BUILD 1284
33.4.2 DMA Client Control Register, DMA_C_CTRL_AUX 1286
33.4.3 DMA Client Channel Select Register, DMA_C_CHAN_AUX 1287
33.4.4 DMA Client Source Address Register, DMA_C_SRC_AUX 1288
33.4.5 DMA Client Destination Address Register, DMA_C_DST_AUX 1289
33.4.6 DMA Client Attributes Register, DMA_C_ATTR_AUX 1290
33.4.7 DMA Client Length Register, DMA_C_LEN_AUX 1293

33.4.8 DMA Client Handle Register, DMA_C_HANDLE_AUX 1294
33.4.9 DMA Event Status Register, DMA_C_EVSTAT_AUX 1295
33.4.10 DMA Client Clear Event Status Register, DMA_C_EVSTAT_CLR_AUX 1296
33.4.11 DMA Client Status Register, DMA_C_STAT_AUX 1297
33.4.12 DMA Client Interrupt Status Register, DMA_C_INTSTAT_AUX 1298
33.4.13 DMA Client Clear Interrupt Status Register, DMA_C_INTSTAT_CLR_AUX 1300
33.4.14 DMA Client Error Handle Register, DMA_C_ERRHANDLE_AUX 1301
33.4.15 DMA Server Control Register, DMA_S_CTRL_AUX 1302
33.4.16 DMA Server Done Status Register, DMA_S_DONESTATD_AUX 1303
33.4.17 DMA Server Clear Done Status Register, DMA_S_DONESTATD_CLR_AUX 1304
33.4.18 DMA Server Channel Tail Pointer Register, DMA_S_TAILC_AUX 1305
33.4.19 DMA Server Channel Middle Pointer Register, DMA_S_MIDC_AUX 1306
33.4.20 DMA Server Channel Head Pointer Register, DMA_S_HEADC_AUX 1307
33.4.21 DMA Server Channel Base Register, DMA_S_BASEC_AUX 1308
33.4.22 DMA Server Channel Last Register, DMA_S_LASTC_AUX 1309
33.4.23 DMA Channel Priority Register, DMA_S_PRIORC_AUX 1310
33.4.24 DMA Channel Control Register, DMA_S_STATC_AUX 1311
33.4.25 CLN_SLV_i_BCR 1312
33.5 Steps to Initiate a DMA Transfer 1312

Chapter 34
User Auxiliary Register Interface 1315

Chapter 35
ROM Patching Unit 1317
35.1 Introduction 1317
35.2 Patch Entry 1318
35.3 Patch Breakpoints 1318
35.4 Patch Jump 1318
35.4.1 BPU 1319
35.4.2 Delay Slots 1319
35.5 Exceptions 1319
35.6 Build-Time Decisions 1319
35.7 Programming the RPU 1319
35.8 Programming Constraints 1320
35.9 Auxiliary Registers 1321
35.9.1 RPU Command Register, RPU_CMND 1322
35.9.2 RPU Address Register, RPU_ADDR 1323
35.9.3 RPU DATA Register, RPU_DATA 1325
35.9.4 RPU Enable Register, RPU_ENABLE 1326
35.9.5 RPU MODE Register, RPU_MODE 1327
35.9.6 RPU Status Register, RPU_STATUS 1328
35.9.7 ROM Base Address Register, RPU_ROM_BASE 1329
35.9.8 RPU Patch Table Base Address Register, RPU_PATCH_TABLE_BASE 1330
35.9.9 RPU Patch Entry Register, PatchTable[n].Address 1331
35.9.10 RPU Build Configuration Register, RPU_BUILD 1332

Part 3
FastMath Extension Pack 1333

Chapter 36
FastMath 1335
36.1 FastMath Data Organization and Addressing 1335
36.1.1 Fractional Data Formats 1335
36.1.2 16-Bit Signed Fractions 1336
36.1.3 32-Bit Signed Fraction (1Q31) 1336
36.2 FastMath Register Set Extensions 1336
36.2.1 Extension Auxiliary Registers 1336
36.2.2 FMP Control Register, FMP_CTRL 1337
36.2.3 Build Configuration Registers 1337
36.2.4 Extension Core Registers 1338
36.3 FastMath Extension Instruction Details 1338
36.3.1 Semantics of FastMath Arithmetic Operations 1339
36.3.2 FastMath fixed-point type declarations 1339
36.3.3 Multiplication of extended-precision 37-bit signed integer values 1341
36.3.4 Rounding of fractional values 1341
36.3.5 Bit-extraction and rounding functions 1342
36.3.6 Support functions for trigonometric operations 1343
36.3.7 Saturating result values to a defined range 1344
36.3.8 Computation of LOG2 (1Q31) 1344
36.3.9 Computation of EXP2 (1Q31) 1346
36.3.10 Computation of COS (1Q31) 1349
36.3.11 Computation of SIN (1Q31) 1350
36.3.12 Computation of ATAN (1Q31) 1351
36.3.13 Computation of SQRT (1Q31) 1352
36.3.14 1Q31 and 1Q15 Versions of SQRT, LOG2, EXP2, SIN, COS and ATAN 1353

Part 4
ARCv3 Floating-Point Unit 1393

Chapter 37
Floating-Point Unit 1395
37.1 Key features of the ARCv3 Floating-point ISA 1395
37.2 Floating-Point Registers 1396
37.3 NaN Formats 1396
37.4 Floating-Point Instruction Encodings 1397
37.4.3 Floating-point Comparison Instructions 1407
37.4.4 Unconditional Single-operand Floating-point Instructions 1408
37.4.5 Conditional Floating-point Move Operations 1408
37.4.6 Floating-point Type Conversion Instructions 1408
37.4.7 Floating-Point Vector/Scalar Moves 1412
37.5 Floating-point MIN and MAX Operation Semantics 1413

Chapter 38
Floating-Point Auxiliary Registers 1415
38.1 Floating-Point Unit Control Register, FP_CTRL 1416
38.2 Floating-Point Unit Status Register, FPU_STATUS 1418
38.3 Floating-Point Unit Vector Status Register, VFPU_STATUS 1422
38.4 Floating-Point Unit File Address Register, FP_RF_ADDR 1424
38.5 Floating-Point Register File Data0 Register, FP_RF_DATA0 1426

38.6 Floating-Point Register File Data 1 Register, FP_RF_DATA1 1427
38.7 Floating-Point Unit Build Register, FPU_BUILD 1428

Chapter 39
Floating-Point Unit Instructions 1429

Part 5
Appendices 1803

Appendix A
Implementation-dependent Behavior 1805
A.1 EM Exception Handling 1805
A.1.1 ECR User mode Setting in the ARC EM Family of Cores 1805
A.2 Cause Codes for Memory Accesses and Data/Address Errors 1805
A.3 Exception Handling 1806
A.3.1 ECR User mode Setting for the ARC HS6x/EV/VPX5 Family of Cores 1806
A.4 Result on Overflow from Integer Division 1806

Appendix B
ARConnect Commands Details 1807
B.1 CMD_CHECK_CORE_ID 1807
B.1.1 Parameter Format 1807
B.1.2 CONNECT_WDATA Register Format 1807
B.1.3 CONNECT_READBACK Register Format 1807
B.2 CMD_INTRPT_GENERATE_IRQ 1808
B.2.1 Parameter Format 1808
B.2.2 CONNECT_WDATA Register Format 1809
B.2.3 CONNECT_READBACK Register Format 1809
B.3 CMD_INTRPT_GENERATE_ACK 1809
B.3.1 Parameter Format 1810
B.3.2 CONNECT_WDATA Register Format 1810
B.3.3 CONNECT_READBACK Register Format 1810
B.4 CMD_INTRPT_READ_STATUS 1810
B.4.1 Parameter Format 1810
B.4.2 CONNECT_WDATA Register Format 1810
B.4.3 CONNECT_READBACK Register Format 1810
B.5 CMD_INTRPT_CHECK_SOURCE 1811
B.5.1 Parameter Format 1811
B.5.2 CONNECT_WDATA Register Format 1811
B.5.3 CONNECT_READBACK Register Format 1811
B.6 CMD_INTRPT_GENERATE_ACK_BIT_MASK 1812
B.6.1 Parameter Format 1812
B.6.2 CONNECT_WDATA Register Format 1812
B.6.3 CONNECT_READBACK Register Format 1813
B.7 CMD_INTRPT_EXT_MODE 1813
B.7.1 Parameter Format 1813
B.7.2 CONNECT_WDATA Register Format 1813
B.7.3 CONNECT_READBACK Register Format 1813
B.8 CMD_INTRPT_SET_PULSE_CNT 1813
B.8.1 Parameter Format 1814

B.8.2 CONNECT_WDATA Register Format . . . 1814
B.8.3 CONNECT_READBACK Register Format . . . 1814
B.9 CMD_INTRPT_READ_PULSE_CNT . . . 1814
B.9.1 Parameter Format . . . 1814
B.9.2 CONNECT_WDATA Register Format . . . 1814
B.9.3 CONNECT_READBACK Register Format . . . 1814
B.10 CMD_SEMA_CLAIM_AND_READ . . . 1814
B.10.1 Parameter Format . . . 1815
B.10.2 CONNECT_WDATA Register Format . . . 1815
B.10.3 CONNECT_READBACK Register Format . . . 1815
B.11 CMD_SEMA_RELEASE . . . 1815
B.11.1 Parameter Format . . . 1815
B.11.2 CONNECT_WDATA Register Format . . . 1815
B.11.3 CONNECT_READBACK Register Format . . . 1816
B.12 CMD_SEMA_FORCE_RELEASE . . . 1816
B.12.1 Parameter Format . . . 1816
B.12.2 CONNECT_WDATA Register Format . . . 1816
B.12.3 CONNECT_READBACK Register Format . . . 1816
B.13 CMD_MSG_SRAM_SET_ADDR . . . 1816
B.13.1 Parameter Format . . . 1816
B.13.2 CONNECT_WDATA Register Format . . . 1817
B.14 CMD_MSG_SRAM_READ_ADDR . . . 1817
B.14.1 Parameter Format . . . 1817
B.14.2 CONNECT_WDATA Register Format . . . 1817
B.14.3 CONNECT_READBACK Register Format . . . 1817
B.15 CMD_MSG_SRAM_SET_ADDR_OFFSET . . . 1817
B.15.1 Parameter Format . . . 1818
B.15.2 CONNECT_WDATA Register Format . . . 1818
B.15.3 CONNECT_READBACK Register Format . . . 1818
B.16 CMD_MSG_SRAM_READ_ADDR_OFFSET . . . 1818
B.16.1 Parameter Format . . . 1818
B.16.2 CONNECT_WDATA Register Format . . . 1818
B.16.3 CONNECT_READBACK Register Format . . . 1819
B.17 CMD_MSG_SRAM_WRITE . . . 1819
B.17.1 Parameter Format . . . 1819
B.17.2 CONNECT_WDATA Register Format . . . 1819
B.17.3 CONNECT_READBACK Register Format . . . 1819
B.18 CMD_MSG_SRAM_WRITE_INC . . . 1819
B.18.1 Parameter Format . . . 1820
B.18.2 CONNECT_WDATA Register Format . . . 1820
B.18.3 CONNECT_READBACK Register Format . . . 1820
B.19 CMD_MSG_SRAM_WRITE_IMM . . . 1820
B.19.1 Parameter Format . . . 1820
B.19.2 CONNECT_WDATA Register Format . . . 1821
B.19.3 CONNECT_READBACK Register Format . . . 1821
B.20 CMD_MSG_SRAM_READ . . . 1821
B.20.1 Parameter Format . . . 1821
B.20.2 CONNECT_WDATA Register Format . . . 1821
B.20.3 CONNECT_READBACK Register Format . . . 1821

B.21 CMD_MSG_SRAM_READ_INC 1821
B.21.1 Parameter Format 1822
B.21.2 CONNECT_WDATA Register Format 1822
B.21.3 CONNECT_READBACK Register Format 1822
B.22 CMD_MSG_SRAM_READ_IMM 1822
B.22.1 Parameter Format 1822
B.22.2 CONNECT_WDATA Register Format 1823
B.22.3 CONNECT_READBACK Register Format 1823
B.23 CMD_MSG_SET_ECC_CTRL 1823
B.23.1 Parameter Format 1823
B.23.2 CONNECT_WDATA Register Format 1823
B.23.3 CONNECT_READBACK Register Format 1824
B.24 CMD_MSG_READ_ECC_CTRL 1824
B.24.1 Parameter Format 1825
B.24.2 CONNECT_WDATA Register Format 1825
B.24.3 CONNECT_READBACK Register Format 1825
B.25 CMD_MSG_READ_ECC_SBE_CNT 1825
B.25.1 Parameter Format 1825
B.25.2 CONNECT_WDATA Register Format 1825
B.25.3 CONNECT_READBACK Register Format 1825
B.26 CMD_MSG_SRAM_CLEAR_ECC_SBE_CNT 1826
B.26.1 Parameter Format 1826
B.26.2 CONNECT_WDATA Register Format 1826
B.26.3 CONNECT_READBACK Register Format 1826
B.27 CMD_DEBUG_RESET 1826
B.27.1 Parameter Format 1826
B.27.2 CONNECT_WDATA Register Format 1826
B.27.3 CONNECT_READBACK Register Format 1827
B.28 CMD_DEBUG_HALT 1827
B.28.1 Parameter Format 1827
B.28.2 CONNECT_WDATA Register Format 1827
B.28.3 CONNECT_READBACK Register Format 1827
B.29 CMD_DEBUG_RUN 1827
B.29.1 Parameter Format 1828
B.29.2 CONNECT_WDATA Register Format 1828
B.29.3 CONNECT_READBACK Register Format 1828
B.30 CMD_DEBUG_SET_MASK 1828
B.30.1 Parameter Format 1828
B.30.2 CONNECT_WDATA Register Format 1829
B.30.3 CONNECT_READBACK Register Format 1829
B.31 CMD_DEBUG_READ_MASK 1829
B.31.1 Parameter Format 1829
B.31.2 CONNECT_WDATA Register Format 1829
B.31.3 CONNECT_READBACK Register Format 1829
B.32 CMD_DEBUG_SET_SELECT 1830
B.32.1 Parameter Format 1830
B.32.2 CONNECT_WDATA Register Format 1830
B.32.3 CONNECT_READBACK Register Format 1830
B.33 CMD_DEBUG_READ_SELECT 1830

B.33.1 Parameter Format . . . 1831
B.33.2 CONNECT_WDATA Register Format . . . 1831
B.33.3 CONNECT_READBACK Register Format . . . 1831
B.34 CMD_DEBUG_READ_EN . . . 1831
B.34.1 Parameter Format . . . 1831
B.34.2 CONNECT_WDATA Register Format . . . 1831
B.34.3 CONNECT_READBACK Register Format . . . 1831
B.35 CMD_DEBUG_READ_CMD . . . 1832
B.35.1 Parameter Format . . . 1832
B.35.2 CONNECT_WDATA Register Format . . . 1832
B.35.3 CONNECT_READBACK Register Format . . . 1832
B.36 CMD_DEBUG_READ_CORE . . . 1832
B.36.1 Parameter Format . . . 1833
B.36.2 CONNECT_WDATA Register Format . . . 1833
B.36.3 CONNECT_READBACK Register Format . . . 1833
B.37 CMD_GFRC_CLEAR . . . 1833
B.37.1 Parameter Format . . . 1834
B.37.2 CONNECT_WDATA Register Format . . . 1834
B.37.3 CONNECT_READBACK Register Format . . . 1834
B.38 CMD_GFRC_READ_LO . . . 1834
B.38.1 Parameter Format . . . 1834
B.38.2 CONNECT_WDATA Register Format . . . 1834
B.38.3 CONNECT_READBACK Register Format . . . 1834
B.39 CMD_GFRC_READ_HI . . . 1834
B.39.1 Parameter Format . . . 1835
B.39.2 CONNECT_WDATA Register Format . . . 1835
B.39.3 CONNECT_READBACK Register Format . . . 1835
B.40 CMD_GFRC_ENABLE . . . 1835
B.40.1 Parameter Format . . . 1835
B.40.2 CONNECT_WDATA Register Format . . . 1835
B.40.3 CONNECT_READBACK Register Format . . . 1835
B.41 CMD_GFRC_DISABLE . . . 1836
B.41.1 Parameter Format . . . 1836
B.41.2 CONNECT_WDATA Register Format . . . 1836
B.41.3 CONNECT_READBACK Register Format . . . 1836
B.42 CMD_GFRC_READ_DISABLE . . . 1836
B.42.1 Parameter Format . . . 1836
B.42.2 CONNECT_WDATA Register Format . . . 1836
B.42.3 CONNECT_READBACK Register Format . . . 1836
B.43 CMD_GFRC_SET_CORE . . . 1837
B.43.1 Parameter Format . . . 1837
B.43.2 CONNECT_WDATA Register Format . . . 1837
B.43.3 CONNECT_READBACK Register Format . . . 1838
B.44 CMD_GFRC_READ_CORE . . . 1838
B.44.1 Parameter Format . . . 1838
B.44.2 CONNECT_WDATA Register Format . . . 1838
B.44.3 CONNECT_READBACK Register Format . . . 1838
B.45 CMD_GFRC_READ_HALT . . . 1839
B.45.1 Parameter Format . . . 1839

B.45.2 CONNECT_WDATA Register Format . . . 1839
B.45.3 CONNECT_READBACK Register Format . . . 1839
B.46 CMD_GFRC_CLK_ENABLE . . . 1839
B.46.1 Parameter Format . . . 1839
B.46.2 CONNECT_WDATA Register Format . . . 1840
B.46.3 CONNECT_READBACK Register Format . . . 1840
B.47 CMD_GFRC_CLK_DISABLE . . . 1840
B.47.1 Parameter Format . . . 1840
B.47.2 CONNECT_WDATA Register Format . . . 1840
B.47.3 CONNECT_READBACK Register Format . . . 1840
B.48 CMD_GFRC_READ_CLK_STATUS . . . 1840
B.48.1 Parameter Format . . . 1840
B.48.2 CONNECT_WDATA Register Format . . . 1840
B.48.3 CONNECT_READBACK Register Format . . . 1840
B.49 CMD_GFRC_READ_FULL . . . 1841
B.49.1 Parameter Format . . . 1841
B.49.2 CONNECT_WDATA Register Format . . . 1841
B.49.3 CONNECT_READBACK_64 Register Format . . . 1841
B.50 CMD_IDU_ENABLE . . . 1841
B.50.1 Parameter Format . . . 1842
B.50.2 CONNECT_WDATA Register Format . . . 1842
B.50.3 CONNECT_READBACK Register Format . . . 1842
B.51 CMD_IDU_DISABLE . . . 1842
B.51.1 Parameter Format . . . 1842
B.51.2 CONNECT_WDATA Register Format . . . 1842
B.51.3 CONNECT_READBACK Register Format . . . 1842
B.52 CMD_IDU_READ_ENABLE . . . 1842
B.52.1 Parameter Format . . . 1842
B.52.2 CONNECT_WDATA Register Format . . . 1842
B.52.3 CONNECT_READBACK Register Format . . . 1843
B.53 CMD_IDU_SET_MODE . . . 1843
B.53.1 Parameter Format . . . 1843
B.53.2 CONNECT_WDATA Register Format . . . 1843
B.53.3 CONNECT_READBACK Register Format . . . 1844
B.54 CMD_IDU_READ_MODE . . . 1844
B.54.1 Parameter Format . . . 1844
B.54.2 CONNECT_WDATA Register Format . . . 1844
B.54.3 CONNECT_READBACK Register Format . . . 1844
B.55 CMD_IDU_SET_DEST . . . 1844
B.55.1 Parameter Format . . . 1845
B.55.2 CONNECT_WDATA Register Format . . . 1845
B.55.3 CONNECT_READBACK Register Format . . . 1845
B.56 CMD_IDU_READ_DEST . . . 1846
B.56.1 Parameter Format . . . 1846
B.56.2 CONNECT_WDATA Register Format . . . 1846
B.56.3 CONNECT_READBACK Register Format . . . 1846
B.57 CMD_IDU_GEN_CIRQ . . . 1846
B.57.1 Parameter Format . . . 1846
B.57.2 CONNECT_WDATA Register Format . . . 1846

B.57.3 CONNECT_READBACK Register Format 1846
B.58 CMD_IDU_ACK_CIRQ 1847
B.58.1 Parameter Format 1847
B.58.2 CONNECT_WDATA Register Format 1847
B.58.3 CONNECT_READBACK Register Format 1848
B.59 CMD_IDU_CHECK_STATUS 1848
B.59.1 Parameter Format 1848
B.59.2 CONNECT_WDATA Register Format 1848
B.59.3 CONNECT_READBACK Register Format 1848
B.60 CMD_IDU_CHECK_SOURCE 1849
B.60.1 Parameter Format 1849
B.60.2 CONNECT_WDATA Register Format 1849
B.60.3 CONNECT_READBACK Register Format 1850
B.61 CMD_IDU_SET_MASK 1850
B.61.1 Parameter Format 1851
B.61.2 CONNECT_WDATA Register Format 1851
B.61.3 CONNECT_READBACK Register Format 1851
B.62 CMD_IDU_READ_MASK 1852
B.62.1 Parameter Format 1852
B.62.2 CONNECT_WDATA Register Format 1852
B.62.3 CONNECT_READBACK Register Format 1852
B.63 CMD_IDU_CHECK_FIRST 1852
B.63.1 Parameter Format 1852
B.63.2 CONNECT_WDATA Register Format 1853
B.63.3 CONNECT_READBACK Register Format 1853
B.64 Examples: ARConnect ICI 1854
B.64.1 Scenario 1 1854
B.64.2 Scenario2 1855
B.65 Examples: ARConnect Semaphore Unit 1857
B.65.1 Scenario 1 1857
B.65.2 Scenario 2 1857
B.66 Examples: ICD 1858
B.66.1 Conditionally Halt Cores Scenario 1858
B.66.2 Selectively Halt Cores Scenario 1859
B.66.3 Selectively Run Cores Scenario 1859
B.66.4 Selectively Reset Cores Scenario 1859
B.67 Examples: GFRC 1861
B.68 Examples: IDU 1861

List of Figures

Figure 4-1 Unified Address Spaces in ARC64 and ARC32 modes 98
Figure 4-2 32-Bit Elements in a 32-bit Operand 100
Figure 4-3 16-Bit Elements in a 32-bit Operand 100
Figure 4-4 Zero Extension of a LIMM in r62. 101
Figure 4-5 Sign Extension of a LIMM in r60 101
Figure 4-6 Expansion of a u6 Literal to a 32-Bit Word (w-shimm) 101
Figure 4-7 Expansion of an s12 Literal to a 32-Bit Word (w-shimm) 101
Figure 4-8 Expansion of a u6 Literal to Half-Word (16-bit, hw-shimm). 101
Figure 4-9 Expansion of an s12 Literal to Half-Word (16-bit, hw-shimm) 101
Figure 4-10 A Half-Word (16-bit) Short Limm Value Replicated to form a 64-Bit Operand. 102
Figure 4-11 A word (32-bit) Short-limm or Limm value Replicated to form a 64-Bit Operand 102
Figure 4-12 Two 16-Bit Vectors with Expanded u6 Operands. 102
Figure 4-13 Two 16-Bit Vectors with Expanded s12 Operands 102
Figure 4-14 Four 8-Bit Vectors with Expanded u6 Operands 102
Figure 4-15 Four 16-Bit Vectors with Expanded s12 Operands 102
Figure 4-16 128-bit Register Data in Byte-Wide Memory, Little-Endian 103
Figure 4-17 Register containing 32-bit Data 104
Figure 4-18 32-bit Register Data in Byte-Wide Memory, Little-Endian 104
Figure 4-19 Register Containing 16-Bit Data. 105
Figure 4-20 16-bit Register Data in Byte-Wide Memory, Little-Endian 105
Figure 4-21 Register Containing 8-Bit Data. 105
Figure 4-22 8-bit Register Data in Byte-Wide Memory 105
Figure 4-23 Register Containing 1-Bit Data. 106

Figure 4-24 Half-word Numbering of a 32-Bit Instruction or Long-immediate Data Item 106
Figure 4-25 Little-endian Offset of each Byte in a 32-Bit Instruction or Immediate 106
Figure 4-26 Big-endian Offset of each Byte in a 32-Bit Instruction or Immediate 106
Figure 4-27 Little-endian Memory Offset of each Byte in a 16-Bit Instruction 107
Figure 4-28 Big-endian Memory Offset of each Byte in a 16-bit Instruction 107
Figure 6-1 ARC64 Core Register Map Summary 115
Figure 6-2 Stack Frame Layout Defined by the ARCv3 ABI 119
Figure 6-3 Layout of Non-volatile Save Area for ARCv3 Prolog and Epilog Instructions 120
Figure 6-4 PCL Register 123
Figure 6-5 IDENTITY Register 128
Figure 6-6 PC Register Addressing Full 64-bit Address Space 129
Figure 6-7 STATUS32 Register 130
Figure 6-8 STATUS32_P0 Register 136
Figure 6-9 AUX_USER_SP Register 137
Figure 6-10 Table 6-1 138
Figure 6-11 INT_VECTOR_BASE Register for ARC64 139
Figure 6-12 AUX_IRQ_ACT Register. 140
Figure 6-13 IRQ_PRIORITY_PENDING Register 141
Figure 6-14 AUX_IRQ_HINT Register. 142
Figure 6-15 IRQ_PRIORITY Register 143
Figure 6-16 JLI_BASE Register when PC_SIZE == 64 144
Figure 6-17 ERET Register 145
Figure 6-18 ERBTA Register 146
Figure 6-19 ERSTATUS Register. 147
Figure 6-20 ECR Register 148
Figure 6-21 EFA Register 149
Figure 6-22 ICAUSE Register 150
Figure 6-23 IRQ_SELECT Register 151
Figure 6-24 IRQ_ENABLE Register 152

Figure 6-25 IRQ_TRIGGER Register 153
Figure 6-26 IRQ_STATUS Register. 154
Figure 6-27 XPU Register 155
Figure 6-28 BTA Register 157
Figure 6-29 IRQ_PULSE_CANCEL Register 158
Figure 6-30 IRQ_PENDING Register. 159
Figure 6-31 User Extension Flags Register, XFLAGS 160
Figure 6-32 BCR_VER Register 162
Figure 6-33 BTA_LINK_BUILD Register. 163
Figure 6-34 VECBASE_AC_BUILD Register for ARC64. 164
Figure 6-35 RF_BUILD Register 165
Figure 6-36 ARC64 MULTIPLY_BUILD Register 167
Figure 6-37 SWAP_BUILD Register. 168
Figure 6-38 NORM_BUILD Register 169
Figure 6-39 MINMAX_BUILD Register. 170
Figure 6-40 BARREL_BUILD Register. 171
Figure 6-41 ISA_CONFIG Build Register 172
Figure 6-42 ISA_CONFIG BCR for Version 0x04 172
Figure 6-43 IRQ_BUILD Register 174
Figure 6-44 ISA_PROFILE Register 176
Figure 6-45 MICRO_ARCH_BUILD Register. 178
Figure 8-1 Long Immediate source data value 232
Figure 8-2 Components of Multi-length Multiplication Operations. 246
Figure 9-1 Branch Conditionally Format. 282
Figure 9-2 Branch Unconditional Far Format. 282
Figure 9-3 Branch on Compare Register-register Format 283
Figure 9-4 Branch on Compare Register-Immediate Format 284
Figure 9-5 Branch and Link Conditionally Format 285
Figure 9-6 Branch and Link Unconditional Far Format 285

Figure 9-7 Load Register with Offset Format 286
Figure 9-8 Store Register with Offset Format 287
Figure 9-9 Store Immediate with Offset Format. 287
Figure 9-10 General Operations Register-Register Format. 289
Figure 9-11 General Operations Register with Unsigned 16-bit Immediate Format 289
Figure 9-12 General Operations Register with Signed 12-bit Immediate Format 289
Figure 9-13 General Operations Conditional Register Format 289
Figure 9-14 General Operations Conditional Register with Unsigned 6-bit Immediate 290
Figure 9-15 Load Register-Register Format 296
Figure 9-16 Extension ALU Operation, register-register 299
Figure 9-17 Extension ALU Operation, register with unsigned 6-bit immediate 300
Figure 9-18 Extension ALU Operation, register with signed 12-bit immediate 300
Figure 9-19 Extension ALU Operation, conditional register 300
Figure 9-20 Extension ALU Operation, cc register with unsigned 6-bit immediate. 300
Figure 10-1 Compact Move Format 305
Figure 10-2 Compact Move Format 306
Figure 10-3 Compact Load and Subtract Format 307
Figure 10-4 Compact Add Format 307
Figure 10-5 Compact Load and Store Format. 308
Figure 10-6 Compact Load Indexed Format 308
Figure 10-7 Load and Add Register-Register Format 309
Figure 10-8 Dual Register with a 3-bit b Register Format. 310
Figure 10-9 Dual Register with a 3-bit Biased Signed Integer Format 310
Figure 10-10 Dual Register with a Implicit LIMM Instructions. 310
Figure 10-11 Code Sequence to Construct a 64-bit Literal using Two LIMMs 311
Figure 10-12 Code sequence to add a 64-bit literal to a GPR using two LIMMs. 311
Figure 10-13 Compact General Operations Register-Register Format 312
Figure 10-14 Compact Single Operand, Jumps, Special Formats 313
Figure 10-15 Compact Zero Operand Instructions 314

Figure 10-16 Compact Load/Store with Offset Format 316
Figure 10-17 Compact Shift/Sub Bit Immediate Format 317
Figure 10-18 Compact Load/Add GP-Relative Format 319
Figure 10-19 Compact Load PCL Relative Format. 320
Figure 10-20 Compact Move Immediate Format 321
Figure 10-21 Compact Branch on Compare Register with Zero Format 322
Figure 10-22 Compact Branch Conditionally Format 323
Figure 10-23 Branch Conditionally with cc Field Format 323
Figure 10-24 Compact Branch and Link Unconditionally Format 325
Figure 10-25 Compact Long-jump and Long-call Format J_S [LIMM] 326
Figure 10-26 Compact Long-jump and Long-call Format JL_S [LIMM]. 326
Figure 10-27 Compact Long-jump and Long-call Format BL_S [LIMM] 326
Figure 11-1 SLEEP Operand 734
Figure 12-1 Example Host Memory Maps, Contiguous Host Memory 863
Figure 12-2 Example Host Memory Maps, Host Memory and Host I/O 863
Figure 13-1 Example 1: MPU_RDB0 Register 871
Figure 13-2 Example 1: MPU_RDP0 Register 872
Figure 13-3 Example 1: MPU_RDB1 Register 872
Figure 13-4 Example 1: MPU_RDP1 Register 872
Figure 13-5 Example 1: MPU_RDB2 Register 872
Figure 13-6 Example 1: MPU_RDP2 Register 872
Figure 13-7 Overlapping Regions Memory Map 873
Figure 13-8 Example 2: MPU_RDB0 Register 873
Figure 13-9 Example 2: MPU_RDP0 Register 873
Figure 13-10 Example 2: MPU_RDB7 Register 873
Figure 13-11 MPU_EN Register 881
Figure 13-12 MPU_ECR Register 884
Figure 13-13 MPU Region Descriptor Base Registers 887
Figure 13-14 MPU Region Descriptor Permissions Registers. 888

Figure 13-15 MPU_MEM_ATTR. 891
Figure 13-16 MPU_BUILD Register 893
Figure 14-1 Stack regions 898
Figure 14-2 Stack Checking Scenarios 899
Figure 14-3 USTACK_TOP 901
Figure 14-4 USTACK_BASE 902
Figure 14-5 KSTACK_TOP 903
Figure 14-6 KSTACK_BASE 904
Figure 14-7 KSTACK_BASE when ADDR_SIZE < 32 904
Figure 14-8 STACK_REGION_BUILD Register 905
Figure 15-1 MMUv48 Page Table Walk with 4KB Minimum Granularity. 908
Figure 15-2 MMUv48 Page Table Walk with 16KB Minimum Granularity. 908
Figure 15-3 MMUv48 Page Table Walk with 64KB Minimum Granularity. 909
Figure 15-4 MMUv48 Level 1 Block Descriptor 909
Figure 15-5 MMUv48 Second level block descriptor 910
Figure 15-6 MMUv48 Third level block descriptor. 910
Figure 15-7 MMUv48-64K fourth level page descriptor 910
Figure 15-8 MMUv48 fourth level page descriptor 911
Figure 15-9 MMUv52 Page Table Walk Overview 912
Figure 15-10 912
Figure 15-11 913
Figure 15-12 913
Figure 15-13 914
Figure 15-14 Block and Page Lower Attributes 914
Figure 15-15 Block and Page Upper Attributes 916
Figure 15-16 Table Attributes 917
Figure 15-17 Two Translation Tables. 920
Figure 15-18 TLB Lookup Flowchart 921
Figure 15-19 Hardware Page Table Walk Flowchart. 922

Figure 15-20 Page Table Walk Algorithm 923
Figure 15-21 MMU_RTP0 Register for MMUv48 924
Figure 15-22 MMU_RTP0 Register for MMUv52 924
Figure 15-23 MMU_RTP1 Register for MMUv48 926
Figure 15-24 MMU_RTP1 Register for MMUv52 926
Figure 15-25 MMU_TLB_IDX Register 928
Figure 15-26 MMU_TLB_CMD Register 930
Figure 15-27 TLB Data Register 932
Figure 15-28 TLB Data Register 933
Figure 15-29 MMU_CTRL Register 934
Figure 15-30 MMU_TTBC Register 935
Figure 15-31 MMU_MEM_ATTR 937
Figure 15-32 MMU_BUILD Register 939
Figure 16-1 AUX_ICCM, base address for ICCM 944
Figure 16-2 ICCM_BUILD Register 945
Figure 17-1 IC_IVIC Register 949
Figure 17-2 IC_CTRL Register 950
Figure 17-3 IC_LIL Register 951
Figure 17-4 IC_IVIR Register 953
Figure 17-5 IC_ENDR Register 955
Figure 17-6 IC_IVIL Register 956
Figure 17-7 IC_RAM_ADDR for Cache Controlled RAM Access (AT==1) 957
Figure 17-8 IC_RAM_ADDR Register for Direct-Cache-Access (AT==0) 957
Figure 17-9 IC_RAM_ADDR for Aliased Configurations 957
Figure 17-10 IC_TAG Register for Reads 960
Figure 17-11 IC_TAG Register for Reads in an Aliased Configuration 960
Figure 17-12 IC_TAG Register for Writes 960
Figure 17-13 IC_DATA Register 964
Figure 17-14 IC_PTAG Register 967

Figure 17-15 IC_WAYLOCK Register 968
Figure 17-16 I_CACHE_BUILD Register 969
Figure 18-1 AUX_DCCM, base address for DCCM 972
Figure 18-2 DCCM_BUILD Register 973
Figure 19-1 DC_IVDC Register 977
Figure 19-2 DC_CTRL Register 979
Figure 19-3 DC_LDL Register 981
Figure 19-4 DC_IVDL Register 982
Figure 19-5 DC_FLSH Register 983
Figure 19-6 DC_FLDL Register 984
Figure 19-7 DC_STARTR Register 985
Figure 19-8 DC_ENDR Register 986
Figure 19-9 DC_RAM_ADDR for Cache Controlled RAM Access (AT==1) 987
Figure 19-10 DC_RAM_ADDR Register for Direct-Cache-Access (AT==0) 987
Figure 19-11 DC_TAG for Reads 990
Figure 19-12 DC_TAG for Writes 990
Figure 19-13 DC_DATA 992
Figure 19-14 D_CACHE_BUILD 993
Figure 20-1 ECC_SBE_COUNT Register 998
Figure 20-2 ERP_CTRL Register 1001
Figure 20-3 ERP_BUILD Register 1003
Figure 21-1 PER_BASE 1008
Figure 21-2 PER_SIZE 1009
Figure 21-3 DMP_PER_BUILD Register 1010
Figure 22-1 WB_CTRL 1012
Figure 22-2 WB_STATUS 1013
Figure 22-3 WB_BUILD 1014
Figure 23-1 BPU_FLUSH Register 1016
Figure 23-2 BPU_CTRL Register 1017

Figure 23-3 BPU_RAM_ADDR Register 1018
Figure 23-4 BC_RAM_INFO Register 1019
Figure 23-5 BC_RAM_BTA Register 1021
Figure 23-6 PT_RAM_DATA Register 1022
Figure 23-7 BPU_BUILD Register 1023
Figure 23-8 Read Access through BPU Debug Controlled Access Mode 1026
Figure 24-1 DEBUG Register 1028
Figure 25-1 Actionpoint Build Register 1038
Figure 25-2 Actionpoint 0 Match Value Register 1038
Figure 25-3 Actionpoint 0 Match Mask Register 1039
Figure 25-4 Actionpoint 0 Control Register 1040
Figure 25-5 Actionpoint 1 Match Value Register 1044
Figure 25-6 Actionpoint 1 Match Mask Register 1045
Figure 25-7 Actionpoint 1 Control Register 1045
Figure 25-8 Actionpoint 2 Match Value Register 1045
Figure 25-9 Actionpoint 2 Match Mask Register 1045
Figure 25-10 Actionpoint 2 Control Register 1046
Figure 25-11 Actionpoint 3 Match Value Register 1046
Figure 25-12 Actionpoint 3 Match Mask Register 1046
Figure 25-13 Actionpoint 3 Control Register 1047
Figure 25-14 Actionpoint 4 Match Value Register 1047
Figure 25-15 Actionpoint 4 Match Mask Register 1047
Figure 25-16 Actionpoint 4 Control Register 1048
Figure 25-17 Actionpoint 5 Match Value Register 1048
Figure 25-18 Actionpoint 5 Match Mask Register 1048
Figure 25-19 Actionpoint 5 Control Register 1048
Figure 25-20 Actionpoint 6 Match Value Register 1049
Figure 25-21 Actionpoint 6 Match Mask Register 1049
Figure 25-22 Actionpoint 6 Control Register 1049

Figure 25-23 Actionpoint 7 Match Value Register 1050
Figure 25-24 Actionpoint 7 Match Mask Register 1050
Figure 25-25 Actionpoint 7 Control Register 1050
Figure 25-26 AP_WP_PC, Watchpoint Program Counter Register 1051
Figure 25-27 SmaRT Operation 1052
Figure 25-28 SMART_BUILD Value. 1053
Figure 25-29 SMART_CONTROL Register 1054
Figure 25-30 SMART_DATA Register 1055
Figure 25-31 SRC_ADDR Value 1055
Figure 25-32 DEST_ADDR Value 1056
Figure 25-33 FLAGS Value Register. 1056
Figure 25-34 DEBUGI Register 1058
Figure 25-35 CLN_SOFTRESET_DEV Register 1060
Figure 25-36 CLN_SOFTRESET_DEV Register 1062
Figure 25-37 SOFT_RESET_PC Register 1063
Figure 25-38 SOFT_RESET_STATUS32 Register 1064
Figure 25-39 SOFT_RESET_STATUS32 Register 1065
Figure 25-40 SOFT_RESET_ECR Register 1067
Figure 25-41 SOFT_RESET_IRQ_ACT Register 1068
Figure 25-42 DB_STATUS Register 1069
Figure 25-43 DB_CMD Register 1070
Figure 25-44 DB_ADDR Register 1070
Figure 25-45 DB_ADDR_HI Register. 1071
Figure 25-46 DB_DATA Register 1071
Figure 25-47 DB_DATA_HI Register 1071
Figure 25-48 DB_RESET Register 1072
Figure 25-49 Debug Interface Timing Diagram 1073
Figure 26-1 CC_INDEX Register. 1078
Figure 26-2 CC_NAME0 Register. 1079

Figure 26-3 CC_NAME1 Register 1080
Figure 26-4 PCT_COUNTL Register 1081
Figure 26-5 PCT_SNAPL Register 1082
Figure 26-6 PCT_CONFIG Register 1083
Figure 26-7 PCT_CONTROL Register 1084
Figure 26-8 PCT_INDEX Register 1086
Figure 26-9 PCT_RANGEL Register 1087
Figure 26-10 PCT_RANGEL Register 1088
Figure 26-11 PCT_UFLAGS Register 1089
Figure 26-12 PCT_INT_CNTL Register 1091
Figure 26-13 PCT_INT_CTRL Register 1092
Figure 26-14 PCT_INT_ACT Register 1093
Figure 26-15 PCT_BUILD Register 1094
Figure 26-16 CC_BUILD Register 1095
Figure 27-1 Interrupt Generated after Timer Reaches Limit Value 1098
Figure 27-2 Timer 0 Count Value Register 1100
Figure 27-3 Timer 0 Control Register 1101
Figure 27-4 Timer 0 Limit Value Register 1103
Figure 27-5 Timer 1 Count Value Register 1104
Figure 27-6 Timer 1 Control Register 1105
Figure 27-7 Timer 1 Limit Value Register 1106
Figure 27-8 AUX_RTC_CTRL Register 1107
Figure 27-9 RTC Count Low Register 1108
Figure 27-10 TIMER_BUILD Register 1109
Figure 28-1 RTT_ADDRESS Register 1112
Figure 28-2 RTT_DATA Register 1113
Figure 28-3 RTT_COMMAND Register 1114
Figure 28-4 RTT_BUILD Register 1115
Figure 28-5 ARCT_BUILD Register 1119

Figure 28-6 ARCT_PRSEL Register 1121
Figure 28-7 ARCT_FLUSH_COMMAND Register 1122
Figure 28-8 ARCT_TSTAMP_ENABLE Register 1123
Figure 28-9 ARCT_DEBUGGER_MESSAGE_ENABLE Register 1124
Figure 28-10 Producer CoreSight Timestamp Register, PR_SYTM_ENABLE. 1125
Figure 28-11 Producer CoreSight Timestamp Register CSTEN_REG_ADDR. 1126
Figure 28-12 Producer Cross Trigger Enable Register, CTIEN_REG_ADDR 1127
Figure 28-13 Producer Trace Attribute Register. 1128
Figure 28-14 Producer Source Enable Register. 1130
Figure 28-15 PR_EVTI_REG Register. 1131
Figure 28-16 RTT_PRDCR_BCR Register 1132
Figure 28-17 Start Address Register. 1134
Figure 28-18 Trace Address Stop Filter Register 1135
Figure 28-19 Producer Type Register. 1136
Figure 28-20 Filter Source Select Register 1137
Figure 28-21 Trace Filter Control Register 1138
Figure 28-22 Producer Watchpoint Status Register 1140
Figure 28-23 Producer Watchpoint Enable Register 1142
Figure 28-24 ARCT_FLUSH_COMMAND Register 1144
Figure 28-25 Producer CoreSight ID Register. 1145
Figure 28-26 Producer ATB SYNC Frame Insertion Register SYNCREQ_REG_ADDR 1146
Figure 28-27 ARCT_OCM_BASE Register 1147
Figure 28-28 ARCT_OCM_SIZE Register 1148
Figure 28-29 ARCT_OCM_WPTR Register. 1149
Figure 28-30 ARCT_TR_STATUS Register 1150
Figure 28-31 ARCT_OFFLOAD_CTRL Register 1152
Figure 28-32 PTRN_GEN Register 1153
Figure 28-33 NEXUS_CLK_DIV Register 1154
Figure 29-1 CLUSTER_BUILD Register. 1156

Figure 29-2 CLN_MST_NOC_0_BCR Register 1162
Figure 29-3 CLN_MST_NOC_1_BCR Register 1163
Figure 29-4 CLN_MST_NOC_2_BCR Register 1164
Figure 29-5 CLN_MST_NOC_3_BCR Register 1165
Figure 29-6 CLN_MST_PER_0_BCR Register 1166
Figure 29-7 CLN_MST_PER_1_BCR Register 1167
Figure 29-8 CLN_MST_CCM_i_BCR Registers 1168
Figure 29-9 CLN_MST_CCM_i_UAUX 1169
Figure 29-10 CLN_MST_CCM_i_XSPC0 1170
Figure 29-11 CLN_MST_CCM_i_XSPC1 1171
Figure 29-12 CLN_SLV_i_BCR Registers 1172
Figure 29-13 CLN_BCR_0 Register 1173
Figure 29-14 CLN_BCR_1 Register 1174
Figure 29-15 CLN_BCR_2 Register 1175
Figure 29-16 CLN_SCM_BCR_0 Register 1176
Figure 29-17 CLN_SCM_BCR_1 Register 1178
Figure 29-18 CLN_SCM_ACR_1 Register 1179
Figure 29-19 CLN_SHMEM_ADDR Register 1180
Figure 29-20 CLN_SHMEM_SIZE Register 1181
Figure 29-21 CLN_CACHE_ADDR_LO0 Register 1182
Figure 29-22 CLN_CACHE_ADDR_LO1 Register 1183
Figure 29-23 CLN_CACHE_ADDR_HI0 Register 1184
Figure 29-24 CLN_CACHE_ADDR_HI1 Register 1185
Figure 29-25 CLN_CACHE_CMD Register 1186
Figure 29-26 CLN_CACHE_STATUS Register 1189
Figure 29-27 CLN_CACHE_ERR Register 1190
Figure 29-28 CLN_CACHE_ERR_ADDR0 1191
Figure 29-29 CLN_CACHE_ERR_ADR1 Register 1192
Figure 29-30 CLN_MST_NOC_i_j_ADDR Register 1193

Figure 29-31 CLN_MST_NOC_i_j_SIZE Register 1194
Figure 29-32 CLN_MST_CCM_i_k_ADDR Register 1195
Figure 29-33 CLN_MST_CCM_i_k_SIZE Register 1195
Figure 29-34 CLN_PERj_BASE Register 1197
Figure 29-35 CLN_PERj_SIZE Register 1198
Figure 29-36 GLOBAL_CLK_GATE_DIS Register 1199
Figure 29-37 CLN_PWR_STATUS_0 Register 1202
Figure 29-38 CLN_PWR_STATUS_1 Register (4 Accelerators Example) 1203
Figure 29-39 CLN_PWR_STATUS_2 Register (4 Accelerators Example) 1204
Figure 29-40 CLN_RTT_PDM_STATUS Register 1205
Figure 29-41 CLN_SLV_i_ARCT Register 1206
Figure 29-42 CLN_QOS_CTL Register 1207
Figure 29-43 CLN_QOS_RDOMi_FOOTPRINT Register 1208
Figure 29-44 CLN_SLV_i_QOS Register 1209
Figure 29-45 CLN_WR_ERR Register 1210
Figure 29-46 CLN_WR_ERR_ADDR0 Register 1211
Figure 29-47 CLN_WR_ERR_ADDR1 Register 1212
Figure 29-48 CLN_ATTN Register 1213
Figure 29-49 CLN_DBANK_ECC_CTRL Register 1214
Figure 29-50 CLN_TBNK_TAG_CTRL Register 1216
Figure 29-51 CLN_STAG_ECC_CTRL Register 1218
Figure 29-52 CLN_CDMA_ECC_CTRL Register 1220
Figure 29-53 CLUSTER_ID Register 1222
Figure 29-54 CPCT_BUILD Register 1224
Figure 29-55 CPCT_CC_NUM 1225
Figure 29-56 CPCT_CC_NAME0 Register 1226
Figure 29-57 CPCT_CC_NAME1 Register 1226
Figure 29-58 CPCT_CC_NAME2 Register 1226
Figure 29-59 CPCT_CC_NAME3 Register 1226

Figure 29-60 CPCT_Index 1227
Figure 29-61 CPCT_CONTROL 1228
Figure 29-62 CPCT_INT_CTRL 1229
Figure 29-63 CPCT_INT_ACT 1230
Figure 29-64 CPCT_<N>_CONFIG 1231
Figure 29-65 CPCT_COUNTL. 1233
Figure 29-66 CPCT_COUNTH 1234
Figure 29-67 CPCT_<N>_SNAPL Register. 1235
Figure 29-68 CPCT_INT_CNTL 1236
Figure 29-69 CPCT_INT_CNTH. 1237
Figure 30-1 HW_PF_CTRL 1242
Figure 31-1 PDM_PSTAT Register 1246
Figure 31-2 RTT_PDM_PSTAT Register 1248
Figure 31-3 PDM_PMODE Register. 1249
Figure 31-4 EXEC_CTRL Register 1251
Figure 31-5 PDM_DVFS_BUILD Register. 1253
Figure 32-1 CONNECT_CMD Register. 1257
Figure 32-2 CONNECT_WDATA Register. 1258
Figure 32-3 CONNECT_READBACK Register 1259
Figure 32-4 CONNECT_READBACK_64 Register 1260
Figure 32-5 CONNECT_SYSTEM_BUILD Register. 1261
Figure 32-6 CONNECT_SEMA_BUILD Register. 1263
Figure 32-7 CONNECT_MESSAGE_BUILD Register 1264
Figure 32-8 CONNECT_IDU_BUILD Register. 1265
Figure 32-9 CONNECT_GFRC_BUILD Register 1266
Figure 32-10 CONNECT_ICI_BUILD Register. 1267
Figure 32-11 CONNECT_ICD_BUILD Register. 1268
Figure 33-1 Cluster DMA Client-Server Architecture. 1271
Figure 33-2 Cluster DMA System Level Architecture. 1273

Figure 33-3 Cluster DMA High-Level Architecture 1274
Figure 33-4 Polling-Mode Sequence Diagram 1277
Figure 33-5 DMA_BUILD Register 1284
Figure 33-6 DMA_C_CTRL_AUX Register 1286
Figure 33-7 DMA_C_CHAN_AUX Register 1287
Figure 33-8 DMA_C_SRC_AUX Register 1288
Figure 33-9 DMA_C_DST_AUX Register 1289
Figure 33-10 DMA_C_ATTR_AUX Register 1290
Figure 33-11 DMA_C_LEN_AUX Register 1293
Figure 33-12 DMA_C_HANDLE_AUX Register 1294
Figure 33-13 DMA_C_EVSTAT_AUX Register 1295
Figure 33-14 DMA_C_EVSTAT_CLR_AUX Register 1296
Figure 33-15 DMA_C_STAT_AUX Register 1297
Figure 33-16 DMA_C_INTSTAT_AUX Register 1298
Figure 33-17 DMA_C_INTSTAT_CLR_AUX Register 1300
Figure 33-18 DMA_C_ERRHANDLE_AUX Register 1301
Figure 33-19 DMA_S_CTRL_AUX Register 1302
Figure 33-20 DMA_S_DONESTATD_AUX Register 1303
Figure 33-21 DMA_S_DONESTATD_AUX Register 1304
Figure 33-22 DMA_S_TAILC_AUX Register 1305
Figure 33-23 DMA_S_MIDC_AUX Register 1306
Figure 33-24 DMA_S_HEADC_AUX Register 1307
Figure 33-25 DMA_S_BASEC_AUX Register 1308
Figure 33-26 DMA_S_LASTC_AUX Register 1309
Figure 33-27 DMA_S_PRIORC_AUX Register 1310
Figure 33-28 DMA_S_STATC_AUX Register 1311
Figure 35-1 RPU Functionality 1317
Figure 35-2 RPU Mapping 1318
Figure 35-3 RPU_CMND Register 1322

Figure 35-4 RPU_ADDR Register 1323
Figure 35-5 RPU_DATA Register 1325
Figure 35-6 RPU_ENABLE Register 1326
Figure 35-7 RPU_MODE Register 1327
Figure 35-8 RPU_STATUS Register 1328
Figure 35-9 RPU_ROM_BASE Register 1329
Figure 35-10 RPU_PATCH_TABLE_BASE Register 1330
Figure 35-11 PATCH_OFFSET_*n* Register 1331
Figure 35-12 RPU_BUILD Register 1332
Figure 36-1 16-Bit Signed 1Q15 Data Representation in a 32-bit Register 1336
Figure 36-2 32-Bit Signed Fractional Data 1336
Figure 36-3 FastMath Extension Pack auxiliary Register Map 1337
Figure 36-4 FMP_BUILD Register 1338
Figure 38-1 FP_CTRL Register 1416
Figure 38-2 FPU_STATUS Register 1418
Figure 38-3 VFPU_STATUS Register 1422
Figure 38-4 FP_RF_ADDR Register 1424
Figure 38-5 FP_RF_DATA0 Register 1426
Figure 38-6 FP_RF_DATA1 Register 1427
Figure 38-7 FPU_BUILD Register 1428
Figure B-1 CONNECT_READBACK Register Format for CMD_CHECK_CORE_ID Command 1807
Figure B-2 CONNECT_READBACK Register Format for CMD_INTRPT_READ_STATUS Command 1810
Figure B-3 CONNECT_READBACK Register Format for CMD_INTRPT_CHECK_SOURCE Command . . . 1811
Figure B-4 CONNECT_READBACK Register Format for CMD_INTRPT_SET_PULSE_CNT Command . . . 1814
Figure B-5 CONNECT_READBACK Register Format for CMD_SEMA_CLAIM_AND_READ Command . . 1815
Figure B-6 CONNECT_WDATA Register Format for CMD_MSG_SRAM_SET_ADDR Command 1817
Figure B-7 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_ADDR Command . . . 1817
Figure B-8 CONNECT_WDATA Register Format for Set-Offset Command 1818
Figure B-9 CONNECT_READBACK Register Format for Read-Offset Command 1819

Figure B-10 CONNECT_WDATA Register Format for CMD_MSG_SRAM_WRITE Command 1819
Figure B-11 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ Command 1821
Figure B-12 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_INC Command 1822
Figure B-13 CONNECT_WDATA Register Format for CMD_MSG_SET_ECC_CTRL Command 1823
Figure B-14 CONNECT_READBACK Register Format for CMD_MSG_READ_ECC_SBE_CNT Command .1825
Figure B-15 CONNECT_WDATA Register Format for CMD_DEBUG_RESET Command 1827
Figure B-16 CONNECT_READBACK Register Format for CMD_DEBUG_READ_MASK Command 1829
Figure B-17 CONNECT_READBACK Register Format for CMD_DEBUG_READ_SELECT Command. 1831
Figure B-18 CONNECT_READBACK Register Format for CMD_DEBUG_READ_EN 1831
Figure B-19 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CMD Command 1832
Figure B-20 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CORE Command. 1833
Figure B-21 CONNECT_READBACK Register Format for CMD_GFRC_READ_LO Command 1834
Figure B-22 CONNECT_READBACK Register Format for CMD_GFRC_READ_HI Command 1835
Figure B-23 CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command 1837
Figure B-24 CONNECT_WDATA Register Format for Command CMD_GFRC_SET_CORE. 1837
Figure B-25 CONNECT_READBACK Register Format for Command CMD_GFRC_READ_CORE 1838
Figure B-26 CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command 1839
Figure B-27 CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command .1841
Figure B-28 CONNECT_READBACK Register Format for CMD_GFRC_READ_FULL Command. 1841
Figure B-29 CONNECT_READBACK Register Format for CMD_IDU_READ_ENABLECommand. 1843
Figure B-30 CONNECT_WDATA Register Format for CMD_IDU_SET_MODE Command. 1843
Figure B-31 CONNECT_WDATA Register Format for CMD_IDU_SET_DEST Command 1845
Figure B-32 CONNECT_READBACK Register Format for CMD_IDU_CHECK_STATUS Command 1848
Figure B-33 CONNECT_READBACK Register Format for CMD_IDU_CHECK_SOURCE command 1850
Figure B-34 CONNECT_WDATA Register Format for CMD_IDU_SET_MASK command. 1851
Figure B-35 CONNECT_READBACK Register Format for CMD_IDU_CHECK_FIRST Command 1853

List of Examples

Example 4-1 Null Instruction Format 108
Example 7-1 Setting Interrupt Priority or Enable (STATUS32.IE) of an Interrupt 186
Example 7-2 Fast Interrupt Entry 190
Example 7-3 Fast Interrupt Exit 190
Example 7-4 Regular Interrupt Entry. 193
Example 7-5 Regular Interrupt Exit 195
Example 7-6 Determine Active Interrupts 197
Example 7-7 Pass Control to an Handler or Perform a Soft Reset. 200
Example 7-8 Exception Vector Code 201
Example 7-9 Exception in a Delay Slot 220
Example 11-1 To obtain a semaphore using EX. 507
Example 11-2 To Release Semaphore using ST 508
Example 11-3 To obtain a semaphore using EX. 510
Example 11-4 To Release Semaphore using ST 511
Example 11-5 Sleep placement in code 735
Example 11-6 Enable Interrupts and Sleep 736
Example 11-7 WEVT placement in code 851
Example 11-8 WLFC Placement in Code. 853
Example 26-1 Example Operations 1089
Example 33-1 Polling-Mode Synchronization 1276
Example 33-2 Interrupt-Mode Synchronization. 1277
Example 33-3 Event-Mode Synchronization. 1278
Example B-1 Segment Program of Scenario 1 — Core1 1854

Example B-2 Segment Program of Scenario 1 — Core2 1854
Example B-3 Segment Program of Scenario 2 - Core0, Core1, Core2 1855
Example B-4 Segment Program of Scenario 2 - Core3 1855
Example B-5 Segment Program of Scenario 1- Core1 1857
Example B-6 Segment Program of Scenario 2 - Core 2 1857
Example B-7 Conditionally Halt Cores Scenario 1858
Example B-8 Selectively Halt Cores Scenario 1859
Example B-9 Selectively Run Cores Scenario 1859
Example B-10 Selectively Reset Cores Scenario 1860
Example B-11 Segment Program of Scenario 1 - Core2 1860
Example B-12 Segment Program to Read Value of GFRC 1861
Example B-13 Segment Program of Common Interrupt 1 Generated by Core 0 1862

List of Tables

Table 1-1 Auxiliary Register List 69
Table 1-2 Build Configuration Registers 75
Table 2-1 Exception Vectors and Cause Codes 79
Table 3-1 ARCv3 Programming Model Configuration Options 91
Table 3-2 Instruction Set Options 92
Table 4-1 Instruction Variants 107
Table 4-2 Operand Description 108
Table 6-1 ARCv3 ARC64 System V ABI Register Usage 116
Table 6-2 Baseline Auxiliary Register Set 124
Table 6-3 Key to Auxiliary Register Access Permissions for LR and SR Instructions 126
Table 6-4 Indexed Table Auxiliary Register (CODE_DENSITY == 1) 126
Table 6-5 XPU, User Extension Permission Auxiliary Register (APEX_OPTION == 1) 127
Table 6-6 Extension Flags Register, XFLAGS 127
Table 6-7 IDENTITY Field Descriptions 128
Table 6-8 STATUS32 Bit-Field Definitions and Read / Write Accessibility 131
Table 6-9 AUX_IRQ_CTRL Field Description 138
Table 6-10 AUX_IRQ_ACT Field Description 140
Table 6-11 IRQ_PRIORITY_PENDING Field Description 141
Table 6-12 IRQ_PRIORITY Field Description 143
Table 6-13 IRQ_SELECT Field Description 151
Table 6-14 IRQ_ENABLE Field Description 152
Table 6-15 IRQ_TRIGGER Field Description 153
Table 6-16 IRQ_STATUS Field Description 154

Table 6-17 IRQ_PULSE_CANCEL Field Description 158
Table 6-18 IRQ_PENDING Field Description. 159
Table 6-19 Build Configuration Registers 161
Table 6-20 BCR_VER Field Descriptions 162
Table 6-21 BTA_LINK_BUILD Field Descriptions. 163
Table 6-22 VECBASE_AC_BUILD Field Descriptions 164
Table 6-23 RF_BUILD Field Descriptions 165
Table 6-24 ARC64 MULTIPLY_BUILD Field Descriptions 167
Table 6-25 SWAP_BUILD Field Descriptions 168
Table 6-26 NORM_BUILD Field Descriptions 169
Table 6-27 MINMAX_BUILD field descriptions 170
Table 6-28 BARREL_BUILD Field Descriptions. 171
Table 6-29 ISA_CONFIG Field Descriptions. 172
Table 6-30 ISA_CONFIG Fields for ARC64 Version 172
Table 6-31 IRQ_BUILD Field Descriptions 174
Table 6-32 ISA_PROFILE Field Descriptions 176
Table 6-33 Product Family and The Release Revisions 178
Table 6-34 MICRO_ARCH_BUILD Field Descriptions 178
Table 7-1 Overview of Privileges in Kernel and User Modes 180
Table 7-2 Interrupt/Exception Vectors 199
Table 7-3 Exception Vectors and Cause Codes. 204
Table 7-4 EFA and ERET Entries for Exception Types 209
Table 7-5 Exception and Interrupt Exit Modes. 219
Table 8-1 Top-level Formats of the Instruction Set 224
Table 8-2 Assignment of Top-level Instruction Formats to Major Opcodes in each Instruction Set Profile 226
Table 8-3 Summary of 32-bit instruction formats. 228
Table 8-4 The F32_FP_OPS Instruction Formats 229
Table 8-5 The F32_FP_MEM Instruction Formats 229
Table 8-6 Summary of 16-bit Instruction Formats 229

Table 8-7 Key for 32-bit Addressing Modes and Encoding Conventions 231
Table 8-8 Key for 16-bit Addressing Modes and Encoding Conventions 231
Table 8-9 Condition Codes 233
Table 8-10 Condition Codes 234
Table 8-11 Load Instruction Encodings 236
Table 8-12 Store Instruction Encodings 237
Table 8-13 Delay Slot Modes 238
Table 8-14 Address Write-Back Modes 239
Table 8-15 Scaling Shift 239
Table 8-16 Cache Bypass Modes 240
Table 8-17 Load Store Data Sizes 240
Table 8-18 Load Data Extend Mode 240
Table 8-19 Instruction Syntax Convention 242
Table 8-20 Integer addition, subtraction and comparison operations 244
Table 8-21 Signed 64x64 MPY producing a 128-bit Result 247
Table 8-22 Unsigned 64x64 MPY producing a 128-bit Result 247
Table 8-23 128x128 MPY producing a 128-bit Result 247
Table 8-24 Signed 128x128 MPY producing a 256-bit Result 248
Table 8-25 Move operations 248
Table 8-26 Bit-wise logical operations 249
Table 8-27 Bit-mask operations 251
Table 8-28 Single Bit Shift and Rotate Operations 251
Table 8-29 Multi-bit Shift Operations 252
Table 8-30 Shift-assist Operations 253
Table 8-31 Selection operations 254
Table 8-32 Byte-swapping operations 254
Table 8-33 Bit-scanning operations 255
Table 8-34 Relational Comparison Operations 255
Table 8-35 Relational Comparison for 64-bit Operations 255

Table 8-36 Integer Multiply, MAC and Divide Operations 256
Table 8-37 Dual and Quad Integer Multiply and MAC Operations 257
Table 8-38 Integer Vector ADD, SUB, MPY operations 259
Table 8-39 Memory Access Instructions 260
Table 8-40 Atomic Memory Operations (AMOs) - 32 bit 264
Table 8-41 Encodings for 32-bit AMO Instructions 264
Table 8-42 ATLD Sub-Opcode Encodings. 265
Table 8-43 Atomic Memory Operations (AMOs) - 64 bit 265
Table 8-44 Encodings for 64-bit AMO Instructions 266
Table 8-45 ATLD Sub-Opcode Encodings. 266
Table 8-46 PREFETCH, PREFETCHW, AND PREALLOC Encodings 267
Table 8-47 Auxiliary Register Operations 267
Table 8-48 Control Flow Operations 268
Table 8-49 Delay Slot Execution Modes. 270
Table 8-50 Branch on Compare/Test Mnemonics 271
Table 8-51 Branch on Compare Pseudo Mnemonics, Register-Register. 272
Table 8-52 Branch on Compare Pseudo Mnemonics, Register-Immediate 273
Table 8-53 Special Instructions 273
Table 8-54 Special Instructions 274
Table 9-1 Hierarchical Sub-formats of the F32_GEN4 Format. 288
Table 9-2 Operand Sub-format Indicators. 288
Table 9-3 Opcode assignment for DOP instructions in major op-code 0x04 (F32_GEN4). 290
Table 9-4 Special DOP sub-formats in F32_GEN4 292
Table 9-5 Opcode assignment for SOP instructions in major op-code 0x04 (F32_GEN4) 293
Table 9-6 Opcode assignment for ZOP instructions in major op-code 0x04 (F32_GEN4) 293
Table 9-7 Opcode assignment for DOP instructions in major op-code 0x0B (F32_GEN_OP64). 294
Table 9-8 Opcode assignment for SOP instructions in major op-code 0x0B (F32_GEN_OP64) 295
Table 9-9 Opcode assignment for ZOP instructions in major op-code 0x0B (F32_GEN_OP64) 295
Table 9-10 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5) 297

Table 9-11 Opcode assignment for SOP instructions in major op-code 0x05 (F32_EXT5) 298
Table 9-12 Opcode assignment for ZOP instructions in major op-code 0x05 (F32_EXT5). 299
Table 9-13 List of FastMath Extension Pack Instructions 301
Table 10-1 16-Bit, LD / ADD Register-Register 309
Table 10-2 Interpretation of the p bit in the ARC64 F16_OP_HREG format 310
Table 10-3 Interpretation of the {j, i} sub-opcode bits in the ARC64 F16_OP_HREG format 310
Table 10-4 DOP_format 16-Bit General Operations. 312
Table 10-5 16-Bit Single Operand Instructions 314
Table 10-6 16-Bit Zero Operand Instructions 315
Table 10-7 Summary of 16-Bit Load and Store Instructions with Offset 316
Table 10-8 16-Bit Shift/SUB/Bit Immediate 317
Table 10-9 Opcodes for Stack-based Operations 318
Table 10-10 16-Bit GP Relative Instructions 319
Table 10-11 16-Bit Branch on Compare 322
Table 10-12 16-Bit Branch, Branch Conditionally. 323
Table 10-13 16-Bit Branch Conditionally 324
Table 10-14 Jump Target Address Calculations for Long Jump/Call Instructions. 326
Table 11-1 List of Instructions 329
Table 11-2 Encodings for 32-bit AMO Instructions 396
Table 11-3 ATLD Sub-Opcode Encodings. 396
Table 11-4 Delay Slot Modes for the Jump and Link Instructions. 528
Table 11-5 Delay Slot Modes for the Jump and Link Instructions. 531
Table 11-6 Delay Slot Modes for the Jump and Link Instructions. 534
Table 11-7 Delay Slot Modes for the Jump and Link Instructions. 537
Table 11-8 PREFETCH Instruction Encodings 661
Table 11-9 Illegal Combinations of <d>, <x>, <aa>, and <zz> for PREFETCH and Load instructions. 662
Table 11-10 SLEEP Operand 734
Table 12-1 Host Accesses to the ARCv3-based processor. 864
Table 12-2 Single Step Flags in Debug Register 865

Table 13-1 Example 1 -- Contiguous Regions Memory Map 871
Table 13-2 Priority Configurations 874
Table 13-3 Setting ECR Register based on EV_ProtV Exception 875
Table 13-4 ECR parameter values for EV_ProtV sources 877
Table 13-5 MPU Register Set 878
Table 13-6 MPU Enable Register 881
Table 13-7 MPU Exception Cause Register 885
Table 13-8 MPU Region Descriptor Base Registers Field Description 887
Table 13-9 MPU Region Descriptor Permission Registers Field Description 888
Table 13-10 Volatile Memory Region 891
Table 13-11 Normal Memory Attributes 892
Table 13-12 AXI4 Attributes 892
Table 13-13 MPU_BUILD Field Description 893
Table 14-1 Stack Checking Specification 899
Table 14-2 Stack Region Checking Auxiliary Registers 900
Table 14-3 STACK_REGION_BUILD Field Descriptions 905
Table 15-1 MMUv48 properties 907
Table 15-2 Access Permission Attributes 915
Table 15-3 Shareability Attributes 915
Table 15-4 APnxt Permissions 917
Table 15-5 Instruction Execution and Data Access Permissions 919
Table 15-6 Special Purpose Registers for TLB Control 923
Table 15-7 MMU_TLB_IDX Addresses 928
Table 15-8 Result Code, Read Only 929
Table 15-9 MMU_TLB_CMD Register Command List 930
Table 15-10 MMU_CTRL Register Field Descriptions 934
Table 15-11 MMU _TTBC Register Field Descriptions 935
Table 15-12 Legal TS0Z and TS1Z 936
Table 15-13 Volatile Memory Region 937

Table 15-14 Normal Memory Attributes 938
Table 15-15 AXI4 Attributes 938
Table 15-16 MMU_BUILD Register Fields 939
Table 16-1 ICCM Registers 943
Table 16-2 ICCM_BUILD Field Descriptions 945
Table 17-1 Instruction Cache Auxiliary Registers 947
Table 17-2 IC_CTRL Control Flags 950
Table 17-3 IC_RAM_ADDRESS Register Behavior 958
Table 17-4 IC_TAG Register Behavior 961
Table 17-5 IC_DATA Register Behavior 965
Table 17-6 I_CACHE_BUILD Field Descriptions 969
Table 18-1 DCCM Auxiliary Registers 971
Table 18-2 DCCM_BUILD Field Descriptions 973
Table 19-1 Data Cache Auxiliary Registers 975
Table 19-2 DC_IVDC Field Descriptions 977
Table 19-3 DC_CTRL Control Flags 979
Table 19-4 DC_FLSH Control Flags 983
Table 19-5 DC_RAM_ADDRESS Register Behavior 988
Table 19-6 DC_TAG Control Flags and Update Conditions 990
Table 19-7 Build Configuration Registers 992
Table 19-8 D_CACHE_BUILD Field Descriptions 993
Table 20-1 Error Protection Auxiliary Registers 997
Table 20-2 ECC_SBE_COUNT Bit Field Description 998
Table 20-3 ERP_CTRL Bit Field Description 1001
Table 20-4 ERP_BUILD Register 1003
Table 21-1 DMP Registers 1007
Table 21-2 DMP_PER_BUILD Field Descriptions 1008
Table 21-3 DMP_PER_BUILD Field Descriptions 1010
Table 22-1 Write Buffer Auxiliary Registers 1011

Table 22-2 WB_CTRL Register Field Description 1012
Table 22-3 WB_STATUS Register Field Description 1013
Table 22-4 WB_BUILD Register Field Description 1014
Table 23-1 BPU Registers 1015
Table 23-2 BPU_CTRL Field Description 1017
Table 23-3 BPU_RAM_ADDR Field Description 1018
Table 23-4 BC_RAM_INFO Field Description 1019
Table 23-5 BPU Branch Types 1020
Table 23-6 BC_RAM_BTA Field Description 1021
Table 23-7 PT_RAM_DATA Bit Field Description. 1022
Table 23-8 BPU_BUILD Field Descriptions. 1023
Table 24-1 Host Debug Interface Auxiliary Registers 1027
Table 24-2 DEBUG Register Field Description 1028
Table 24-3 Internal Clock During Sleep Mode 1031
Table 25-1 Auxiliary Registers for Actionpoints 1034
Table 25-2 State of Auxiliary Registers upon Reset 1036
Table 25-3 Actionpoint Build Field Descriptions 1038
Table 25-4 Actionpoint Target Field, AT, Type 1040
Table 25-5 Transaction Type Field, TT, Type 1041
Table 25-6 Mode Field, M, Type 1041
Table 25-7 Pair Field, P, Type 1042
Table 25-8 Actionpoint Action Field, AA Field, Type 1043
Table 25-9 Quad Type 1044
Table 25-10 SMART_BUILD 1053
Table 25-11 SMART_CONTROL Register. 1054
Table 25-12 SMART_DATA Register 1055
Table 25-13 SRC_ADDR Value 1056
Table 25-14 DEST_ADDR Register. 1056
Table 25-15 FLAGS Value Register. 1057

Table 25-16 DEBUGI Register Field Description 1058
Table 25-17 CLN_SOFTRESET_CORES 1060
Table 25-18 CLN_SOFTRESET_DEV 1062
Table 25-19 SOFT_RESET_STATE 1065
Table 25-20 Debug Register Addresses 1069
Table 25-21 DB_CMD Field Description 1070
Table 26-1 Performance Auxiliary Registers 1076
Table 26-2 Countable-Conditions Registers 1076
Table 26-3 CC_INDEX Field Description 1078
Table 26-4 CC_NAME0 Field Description 1079
Table 26-5 CC_NAME1 Field Description 1080
Table 26-6 PCT_CONFIG Field Description 1083
Table 26-7 PCT_CONTROL Field Description 1084
Table 26-8 PCT_INDEX Field Description 1086
Table 26-9 PCT_UFLAGS Field Description 1090
Table 26-10 PCT_BUILD Field Description 1094
Table 26-11 CC_BUILD Field Description 1095
Table 27-1 Auxiliary Registers for Hardware Counter 0 (has_timer_0 == true) 1098
Table 27-2 Auxiliary Registers for Hardware Counter 1 (has_timer_1 == true) 1098
Table 27-3 Auxiliary Register for Real-time counter 1099
Table 27-4 TIMER 0 Field Descriptions 1101
Table 27-5 RTC Control Register 1107
Table 27-6 TIMER_BUILD Field Descriptions 1109
Table 28-1 ARC Trace Registers 1111
Table 28-2 RTT_COMMAND Bit Field Description 1114
Table 28-3 RTT_BUILD Field Descriptions 1115
Table 28-4 ARC Trace Address Map 1116
Table 28-5 ARC Trace Producer Registers 1116
Table 28-6 ARCT_BUILD Field Descriptions 1119

Table 28-7 ARCT_PRSEL Field Descriptions 1121
Table 28-8 ARCT_FLUSH_COMMAND Field Descriptions 1122
Table 28-9 ARCT_TSTAMP_ENABLE Field Descriptions 1123
Table 28-10 ARCT_DEBUGGER_MESSAGE_ENABLE Field Descriptions 1124
Table 28-11 Producer Cross Trigger Enable Register PR_SYTM_ENABLE Description 1125
Table 28-12 Producer Cross Trigger Enable Register CSTSEN_REG_ADDR Description 1126
Table 28-13 Producer Cross Trigger Enable Register Description. 1127
Table 28-14 Producer Trace Attribute Register Description. 1128
Table 28-15 Producer Source Enable Description 1130
Table 28-16 PR_EVTI_REG Field Descriptions. 1131
Table 28-17 RTT_PRDCR_BCR Description 1132
Table 28-18 Start Address Register and Stop Register Address Offsets 1133
Table 28-19 Start Address Register Field Descriptions 1134
Table 28-20 Stop Address Register Field Descriptions 1135
Table 28-21 Producer Type Register Field Descriptions 1136
Table 28-22 Filter Source Select Register Description 1137
Table 28-23 Trace Filter Control Register 1138
Table 28-24 Producer Source Watchpoint Status Register Description 1140
Table 28-25 Producer Watchpoint Enable Register Description 1142
Table 28-26 ARC Trace Control Registers 1143
Table 28-27 ARCT_FLUSH_COMMAND Field Descriptions 1144
Table 28-28 Producer CoreSight ID Register Description. 1145
Table 28-29 Producer SYNC Frame Insertion Register Description 1146
Table 28-30 ARCT_OCM_BASE Field Descriptions 1147
Table 28-31 ARCT_OCM_SIZE Field Descriptions 1148
Table 28-32 ARCT_OCM_WPTR Field Descriptions. 1149
Table 28-33 ARCT_TR_STATUS Field Descriptions 1150
Table 28-34 ARCT_OFFLOAD_CTRL Field Descriptions 1152
Table 28-35 PTRN_GEN Field Descriptions 1153

Table 28-36 NEXUS_CLK_DIV Field Descriptions 1154
Table 29-1 CLUSTER_BUILD Field Description 1156
Table 29-2 Cluster Network Register Overview 1156
Table 29-3 CLN Status and Control Register Overview 1157
Table 29-4 CLN_MST_NOC_0_BCR Field Descriptions 1162
Table 29-5 CLN_MST_NOC_1_BCR Field Description 1163
Table 29-6 CLN_MST_NOC_2_BCR Field Description 1164
Table 29-7 CLN_MST_NOC_3_BCR Field Description 1165
Table 29-8 CLN_MST_PER_0_BCR Field Description 1166
Table 29-9 CLN_MST_PER_1_BCR Field Description 1167
Table 29-10 CLN_MST_CCM_i_BCR Field Description 1168
Table 29-11 CLN_MST_CCM_i_UAUX Field Description 1169
Table 29-12 CLN_MST_CCM_i_XSPC0 Field Description 1170
Table 29-13 CLN_MST_CCM_i_XPSC1 Field Description 1171
Table 29-14 CLN_SLV_i_BCR Field Description 1172
Table 29-15 CLN_BCR_0 Field Description 1173
Table 29-16 CLN_BCR_1 Field Description 1174
Table 29-17 CLN_BCR_1 Field Description 1175
Table 29-18 CLN_SCM_BCR_0 Field Description 1176
Table 29-19 CLN_SCM_BCR_1 Field Description 1178
Table 29-20 CLN_SCM_ACR_1 Field Description 1179
Table 29-21 CLN_SHMEM_ADDR Field Description 1180
Table 29-22 CLN_SHMEM_SIZE Field Description 1181
Table 29-23 CLN_CACHE_ADDR_LO0 Register 1182
Table 29-24 CLN_CACHE_ADDR_LO1 Field Description 1183
Table 29-25 CLN_CACHE_ADDR_HI0 Field Description 1184
Table 29-26 CLN_CACHE_ADDR_HI1 Field Description 1185
Table 29-27 CLN_CACHE_CMD Field Description 1186
Table 29-28 CLN_CACHE_STATUS Field Description 1189

Table 29-29 CLN_CACHE_ERR Field Description 1190
Table 29-30 CLN_CACHE_ERR_ADDR0 Field Description 1191
Table 29-31 CLN_CACHE_ ERR _ADR1 Description 1192
Table 29-32 CLN_MST_NOC_i_j_ADDR Field Description 1193
Table 29-33 CLN_MST_NOC_i_j_SIZE Field Description 1194
Table 29-34 CLN_MST_CCM_i_k_ADDR Field Description 1195
Table 29-35 CLN_MST_CCM_i_k_SIZE Field Description 1196
Table 29-36 CLN_PERj_BASE Field Description 1197
Table 29-37 CLN_PERj_SIZE Field Description 1198
Table 29-38 GLOBAL_CLK_GATE_DIS Register 1199
Table 29-39 CLN_PWR_STATUS_0 Field Description 1202
Table 29-40 CLN_PWR_STATUS_1 Field Description 1203
Table 29-41 CLN_PWR_STATUS_2 Field Description 1204
Table 29-42 CLN_RTT_PDM_STATUS Field Description 1205
Table 29-43 CLN_SLV_i_ARCT Field Description 1206
Table 29-44 CLN_QOS_CTL Field Description 1207
Table 29-45 CLN_QOS_RDOMi_FOOTPRINT Field Description 1208
Table 29-46 CLN_SLV_i_QOS Field Description 1209
Table 29-47 CLN_WR_ERR Field Description 1210
Table 29-48 CLN_WR_ERR_ADDR0 Field Description 1211
Table 29-49 CLN_WR_ERR_ADDR1 Field Description 1212
Table 29-50 CLN_ATTN Field Description 1213
Table 29-51 CLN_DBANK_ECC_CTRL Field Description 1215
Table 29-52 CLN_TBNK_ECC_CTRL Field Description 1217
Table 29-53 CLN_STAG_ECC_CTRL Field Description 1218
Table 29-54 CLN_CDMA_ECC_CTRL Field Description 1220
Table 29-55 Cluster Registers 1221
Table 29-56 CLUSTER_ID Register 1222
Table 29-57 Cluster Performance Counter Registers 1222

Table 29-58 CPCT_BUILD Register Field Description 1224
Table 29-59 CPCT_CC_NUM Register Field Description 1225
Table 29-60 CPCT_INDEX Register field description 1227
Table 29-61 CPCT_CONTROL Register Field Descrciption 1228
Table 29-62 CPCT_INT_CTRL Register Field Description 1229
Table 29-63 CPCT_INT_ACT Register Field Description 1230
Table 29-64 CPCT_<N>_CONFIG Register field description 1231
Table 30-1 HW_PF_CTRL Field Description 1242
Table 31-1 Power Domain Management Registers 1245
Table 31-2 PDM_PSTAT Field Description 1246
Table 31-3 RTT_PDM_PSTAT Field Description 1248
Table 31-4 PDM_PMODE Field Description 1249
Table 31-5 Execution Control Registers 1250
Table 31-6 EXEC_CTRL Field Description 1251
Table 31-7 PDM_DVFS_BUILD Field Description 1253
Table 32-1 ARConnect Registers 1255
Table 32-2 CONNECT_CMD Register Field Description 1257
Table 32-3 CONNECT_SEMA_BUILD Register Field Description 1263
Table 32-4 CONNECT_MESSAGE_BUILD Register Field Description 1264
Table 32-5 CONNECT_IDU_BUILD Field Description 1265
Table 32-6 CONNECT_GFRC_BUILD Field Description 1266
Table 32-7 CONNECT_ICI_BUILD Field Description 1267
Table 32-8 CONNECT_ICD_BUILD Field Description 1268
Table 33-1 Cluster DMA Auxiliary Register List 1282
Table 33-2 DMA_BUILD Field Description 1284
Table 33-3 DMA_C_CTRL_AUX Field Description 1286
Table 33-4 DMA_C_CHAN_AUX Field Description 1287
Table 33-5 DMA_C_SRC_AUX Field Description 1288
Table 33-6 DMA_C_DST_AUX Field Description 1289

Table 33-7 DMA_C_ATTR_AUX Field Description 1290
Table 33-8 DMA_C_LEN_AUX Field Description. 1293
Table 33-9 DMA_C_HANDLE_AUX Field Description 1294
Table 33-10 DMA_C_EVSTAT_AUX Field Description 1295
Table 33-11 DMA_C_EVSTAT_CLR_AUX Field Description 1296
Table 33-12 DMA_C_STAT_AUX Field Description. 1297
Table 33-13 DMA_C_INTSTAT_AUX Field Description 1298
Table 33-14 DMA_C_INTSTAT_CLR_AUX Field Description 1300
Table 33-15 DMA_C_ERRHANDLE_AUX Field Description 1301
Table 33-16 DMA_S_CTRL_AUX Field Description 1302
Table 33-17 DMA_S_DONESTATD_AUX Field Description. 1303
Table 33-18 DMA_S_DONESTATD_AUX Field Description. 1304
Table 33-19 DMA_S_STATC_AUX Field Description. 1311
Table 35-1 RPU Indirect Programming Interface Registers 1321
Table 35-2 RPU_CMND Bit Field Description 1322
Table 35-3 RPU Programming Registers 1323
Table 35-4 RPU_BUILD Field Descriptions 1332
Table 36-1 FMP Auxiliary Registers. 1336
Table 36-2 FMP_CTRL Field Descriptions 1337
Table 36-3 Build Configuration Registers 1337
Table 36-4 FMP_BUILD Field Descriptions 1338
Table 37-1 The F32_FP_OPS Instruction Formats 1397
Table 37-2 The F32_FP_MEM Instruction Formats 1397
Table 37-3 Interpretation and Legality of <P> Field 1398
Table 37-4 Interpretation of <ZZ> bits in FP Load/Store Instructions. 1399
Table 37-5 Supported Memory Data Transfer Sizes as a Function of Core Configuration 1400
Table 37-6 Supported Memory Data Transfer Sizes as a Function of Core Configuration 1400
Table 37-7 Negation of operands for fusedMultiplyAdd operations. 1402
Table 37-8 Triple-operand Floating-point Instructions. 1402

Table 37-9 FPR_WIDTH and VFP_WIDTH Definitions 1403
Table 37-10 Enumerated values of VFP_MAX_VLEN as a function of configuration 1403
Table 37-11 Dual-operand Floating-point Instructions 1404
Table 37-12 Summary of Vector Permutation Instructions and Their Semantics 1406
Table 37-13 Enumerated Mappings, from Source to Destination, for all Vector Permutation Instructions . . . 1407
Table 37-14 Unconditional Single-Operand Floating-point Instructions 1408
Table 37-15 Conditional Floating-point Instructions 1408
Table 37-16 Floating-point Conversion Instructions Enumerated 1409
Table 37-17 Summary of FMV* Instructions and their Semantics 1411
Table 37-18 FP Scalar Insertion, Extraction, and Replication 1412
Table 38-1 FPU Registers 1415
Table 38-2 FP_CTRL Field Description 1416
Table 38-3 FPU_STATUS Field Description 1418
Table 38-4 VFPU_STATUS Field Description 1422
Table 38-5 FP_RF_ADDR Description 1424
Table 38-6 FPU_BUILD Field Description 1428
Table 39-1 Floating-Point Instructions 1429
Table 39-2 Permutation Instruction Result 1689
Table 39-3 Permutation Instruction Result 1691
Table 39-4 Permutation Instruction Result 1693
Table 39-5 Permutation Instruction Result 1695
Table 39-6 Permutation Instruction Result 1697
Table 39-7 Permutation Instruction Result 1699
Table 39-8 Permutation Instruction Result 1701
Table 39-9 Permutation Instruction Result 1703
Table 39-10 Permutation Instruction Result 1705
Table 39-11 Permutation Instruction Result 1707
Table 39-12 Permutation Instruction Result 1709
Table 39-13 Permutation Instruction Result 1711

Table 39-14 Permutation Instruction Result 1713
Table 39-15 Permutation Instruction Result 1715
Table 39-16 Permutation Instruction Result 1717
Table 39-17 Permutation Instruction Result 1719
Table 39-18 Permutation Instruction Result 1721
Table 39-19 Permutation Instruction Result 1723
Table 39-20 Permutation Instruction Result 1725
Table 39-21 Permutation Instruction Result 1727
Table 39-22 Permutation Instruction Result 1729
Table A-1 Implementation Behavior on Division Overflow 1806
Table B-1 CONNECT_READBACK Register Field Descriptions for CMD_CHECK_CORE_ID 1808
Table B-2 CONNECT_READBACK Register Field Descriptions for CMD_INTERRUPT_READ_STATUS . . 1811
Table B-3 CONNECT_READBACK Register Field Descriptions for CMD_INTRPT_CHECK_SOURCE 1812
Table B-4 CONNECT_READBACK Register Field Descriptions for CMD_SEMA_CLAIM_AND_READ . . . 1815
Table B-5 CONNECT_WDATA Register Field Description for CMD_MSG_SET_ECC_CTRL 1824
Table B-6 CONNECT_READBACK Register Field Description for CMD_MSG_READ_ECC_SBE_CNT 1825
Table B-7 CONNECT_WDATA Field Descriptions for CMD_DEBUG_RESET 1827
Table B-8 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_EN 1832
Table B-9 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_CORE. 1833
Table B-10 CONNECT_WDATA Field Descriptions. 1837
Table B-11 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MODE. 1843
Table B-12 CONNECT_WDATA Register Field Description for CMD_IDU_SET_DEST 1845
Table B-13 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_STATUS 1848
Table B-14 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_SOURCE. 1850
Table B-15 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MASK 1851
Table B-16 CONNECT_WDATA Register Field Description for CMD_IDU_CHECK_FIRST 1853

Preface

Note for Readers

The product name *DesignWare Cores* is in the process of being updated to *Synopsys IP*.

Web Resources

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

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

Accessing SolvNetPlus

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

To access SolvNetPlus:

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

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

Customer Support

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

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

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

Contacting the Synopsys Technical Support Center

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

- *For the fastest response*, enter a case through SolvNetPlus:
  a. https://solvnetplus.synopsys.com

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

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

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

SolvNetPlus Online Customer Support

- When you send an e-mail message to support_center@synopsys.com, your email is queued and then, on a first-come, first-served basis, manually routed to the correct support engineer. Kindly note to:
  - Include the full product name and version number in your email so it can be routed correctly.
  - For simulation issues, include the timestamp of any signals or any locations in waveforms that are not understood.
  - Attach any debug files you created.
- Or, telephone your local support center:
  - North America:

    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:

    https://www.synopsys.com/support/global-support-centers.html

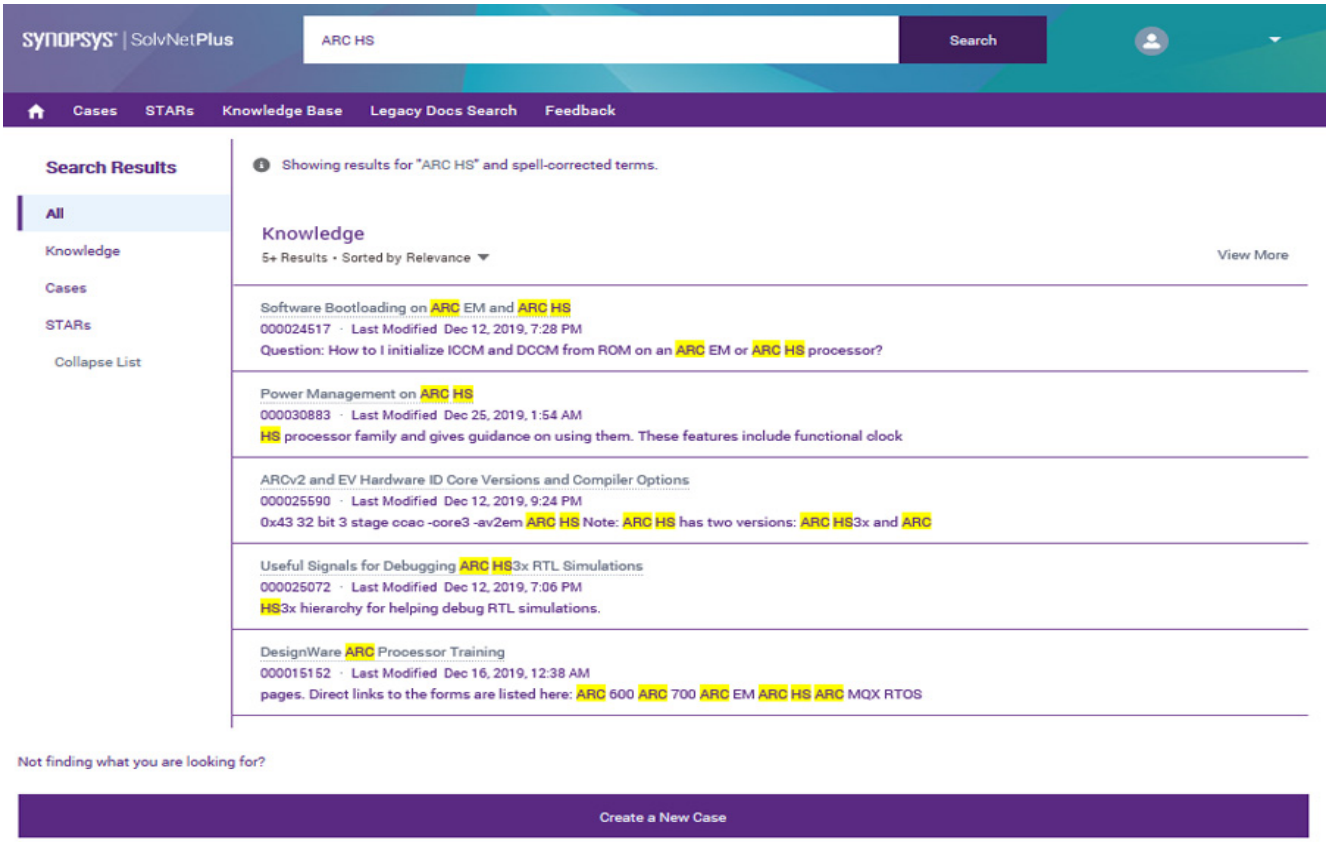
Accessing Application Notes

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

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

The figure below shows example results.

Note Application notes are updated frequently, so results can vary.



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.

Note to Readers

Note For ARCv3 documentation, ARConnect occurrences stand for MCIP usage.

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
5820-001	August 2024	■ Initial version

Quick Reference

In this part:

- Build and Auxiliary Register List
- Exceptions List

1 Build and Auxiliary Register List

This chapter lists all the auxiliary registers and build configuration registers available in the ARCv3 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
0x04	Core Identity Register, IDENTITY	PD1
0x05	Debug Register, DEBUG	PD1
0x06	Program Counter, PC	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
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

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x1D	Instruction Cache Data Access, IC_DATA	PD1
0x1E	Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG	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
0x4F	Hardware Prefetcher Control Auxiliary Register, HW_PF_CTRL	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
0x60 -0x7F 0xC0 -0xFF	Build Configuration Registers	PD1
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
0x104	RTC Count Register, AUX_RTC_LOW	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

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
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
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
0x252	Snapshot-Value Registers, PCT_SNAPL	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
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
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
0x26A	Private Peripheral Aperture Base Address, PER_BASE	PD1
0x26B	Private Peripheral Aperture Size, PER_SIZE	PD1
0x298	CLUSTER ID Register, CLUSTER_ID	PD1
0x300	Floating-Point Unit Control Register, FP_CTRL	PD1
0x301	Floating-Point Unit Status Register, FPU_STATUS	PD1
0x310	Floating-Point Unit File Address Register, FP_RF_ADDR	PD1
0x311	Floating-Point Register File Data0 Register, FP_RF_DATA0	PD1
0x312	Floating-Point Register File Data 1 Register, FP_RF_DATA1	PD1
0x380	ARC Trace Address Register, RTT_ADDRESS	PD1
0x381	ARC Trace DATA Register, RTT_DATA	PD1
0x382	ARC Trace CMD Register, RTT_COMMAND	PD1
0x390	RPU Command Register, RPU_CMND	PD1
0x391	RPU Address Register, RPU_ADDR	PD1
0x392	RPU DATA Register, RPU_DATA	PD1
0x3A0	Soft Reset PC, SOFT_RESET_PC	
0x3A1	Soft Reset STATUS32, SOFT_RESET_STATUS32	
0x3A2	Soft Reset State, SOFT_RESET_STATE	
0x3A3	Soft Reset Saved ECR, SOFT_RESET_ECR	

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x3A4	Soft Reset Save Active Interrupt Register, SOFT_RESET_IRQ_ACT	
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
0x442	MPU Attribute Register, MPU_MEM_ATTR	PD1
0x44F	User Extension Flags Register, XFLAGS	PD1
0x460	MMU Root Translation Pointer0 Auxiliary Register, MMU_RTP0	PD1
0x462	MMU Root Translation Pointer1 Auxiliary Register, MMU_RTP1	PD1
0x464	MMU TLB Index Register, MMU_TLB_IDX	PD1
0x465	MMU TLB Command Register, MMU_TLB_CMD	PD1
0x466	MMU TLB Data 0 Register, MMU_TLB_DATA0	PD1
0x467	MMU TLB Data 1 Register, MMU_TLB_DATA1	PD1
0x468	MMU Control Register, MMU_CTRL	PD1
0x469	MMU Translation Table Base Control Register, MMU_TTBC	PD1
0x46A	MMU Attribute Register, MMU_MEM_ATTR	PD1
0x470	FMP Control Register, FMP_CTRL	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0x480	Flush and Initialize Branch Predictor Register, BPU_FLUSH	PD1
0x481	BPU Control Register, BPU_CTRL	PD1
0x600	ARConnect Command Register, CONNECT_CMD	PD1
0x601	ARConnect Write Data Register, CONNECT_WDATA	PD1
0x602	ARConnect Read Data Register, CONNECT_READBACK	PD1
0x603	ARConnect Read 64-Bit Data Register, CONNECT_READBACK _64	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
0x700	SmaRT Control Register, SMART_CONTROL	PD1
0x701	SmaRT Data Register, SMART_DATA	PD1
0x9A9	Global Clock Disable Register, GLOBAL_CLK_GATE_DIS	Always on
0xD00	DMA Client Control Register, DMA_C_CTRL_AUX	PD1
0xD01	DMA Client Channel Select Register, DMA_C_CHAN_AUX	PD1
0xD02	DMA Client Source Address Register, DMA_C_SRC_AUX	PD1
0xD04	DMA Client Destination Address Register, DMA_C_DST_AUX	PD1
0xD06	DMA Client Attributes Register, DMA_C_ATTR_AUX	PD1
0xD07	DMA Client Length Register, DMA_C_LEN_AUX	PD1
0xD08	DMA Client Handle Register, DMA_C_HANDLE_AUX	PD1
0xD0A	DMA Event Status Register, DMA_C_EVSTAT_AUX	PD1
0xD0B	DMA Client Clear Event Status Register, DMA_C_EVSTAT_CLR_AUX	PD1
0xD0C	DMA Client Status Register, DMA_C_STAT_AUX	PD1
0xD0D	DMA Client Interrupt Status Register, DMA_C_INTSTAT_AUX	PD1
0xD0E	DMA Client Clear Interrupt Status Register, DMA_C_INTSTAT_CLR_AUX	PD1
0xD0F	DMA Client Error Handle Register, DMA_C_ERRHANDLE_AUX	PD1
0xD10	DMA Server Control Register, DMA_S_CTRL_AUX	PD1
0xD20+D	DMA_S_DONESTATD_AUX	PD1
0xD40+D	DMA Server Clear Done Status Register, DMA_S_DONESTATD_CLR_AUX	PD1
0xD80+(C*8)	DMA Server Channel Tail Pointer Register, DMA_S_TAILC_AUX	PD1

Table 1-1 Auxiliary Register List (Continued)

Register Address	Register Name	Power Domain
0xD81+(C*8)	DMA Server Channel Middle Pointer Register, DMA_S_MIDC_AUX	PD1
0xD82+(C*8)	DMA Server Channel Head Pointer Register, DMA_S_HEADC_AUX	PD1
0xD83+(C*8)	DMA Server Channel Base Register, DMA_S_BASEC_AUX	PD1
0xD84+(C*8)	DMA Server Channel Last Register, DMA_S_LASTC_AUX	PD1
0xD85+(C*8)	DMA Channel Priority Register, DMA_S_PRIORC_AUX	PD1
0xD86+(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 ARCv3-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 ARCv3 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 ARCv3-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

Table 1-2 Build Configuration Registers (Continued)

Number	Name	Power Domain
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
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
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_PER_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
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
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
0xE6	DMA Build Configuration Register, DMA_BUILD	PD1
0xE8	FMP Build Configuration Register, FMP_BUILD	PD1
0xF2	ARC Trace Build Configuration Register, RTT_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	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	Para-meter	
Reset	Reset	0x00	0x00	0x00	0x00	0x000000
Bus Error from Instruction Memory	Memory Error	0x04	0x01	0x00	0x0u	0x01000u(a)
Bus Error from Data Memory	Memory Error	0x04	0x01	0x10	0x0u	0x01100u(a)(b)
Bus Error from Data Memory during a Hardware table walk	Memory Error	0x04	0x01	0x41	0x0u	0x01410u(a)
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
Data access spanning multiple data memory targets	Memory Error	0x04	0x01	0x12	0x00	0x011200

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
Cached LD/ST instruction is attempted on a peripheral interface	Memory Error	0x04	0x01	0x15	0x00	0x011500
Memory error on Data Access	Memory Error	0x04	0x01	0x16	0x00	0x011600
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)
Non-Maskable imprecise exception	EV_MachineCheck	0x0C	0x03	0x14	0x00	0x031400
Non-maskable imprecise exception double fault	EV_MachineCheck	0x0C	0x03	0x14	0x01	0x031401
Machine check - uncorrectable ECC or parity error in vector memory	EV_MachineCheck	0x0C	0x03	0x80	0x00	0x038000
Translation fault exception on Instruction Fetch	EV_IMMUFault	0x10	0x04	0x00	0x00	0x040000

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
Translation fault exception on Invalid Instruction Address	EV_IMMUFault	0x10	0x04	0x08	0x00	0x040800
Access flag exception on Instruction Fetch	EV_IMMUFault	0x10	0x04	0x10	0x00	0x041000
Illegal Instruction Fetch ICCM translation	EV_IMMUFault	0x10	0x04	0x20	0x00	0x042000
Translation fault exception on Data Memory Read	EV_DMMUFault	0x14	0x05	0x01	0x00	0x050100
Translation fault exception on Data Memory Write	EV_DMMUFault	0x14	0x05	0x02	0x00	0x050200
Translation fault exception on Data Memory Read-Modify-Write (EX/AMOs)	EV_DMMUFault	0x14	0x05	0x03	0x00	0x050300
Translation fault exception on Invalid Data Address	EV_DMMUFault	0x14	0x05	0x08	0x00	0x050800
Access flag exception on Data Access	EV_DMMUFault	0x14	0x05	0x10	0x00	0x051000
Illegal Data Access ICCM translation	EV_DMMUFault	0x14	0x05	0x20	0x00	0x052000
Illegal Data Access DCCM translation	EV_DMMUFault	0x14	0x05	0x30	0x00	0x053000
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

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
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
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

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
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
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(c)
Invalid Operation, Floating-point extension exceptions	EV_Extension	0x28	0x0A	0x00	0x01	0x0A0001

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

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
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
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 216.

(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

nn = 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
ARCv3 Baseline ISA

In this part:

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

3 Introduction

This document is intended for programmers of the ARCv3 ISA (Instruction-Set Architecture). The ARCv3 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 ARCv3 ISA.

The ARCv3 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 ARCv3 ISA, compact 16-bit encodings of frequently used 32-bit instructions are provided. These can be freely intermixed with 32-bit encodings.

3.1 Typographic Conventions

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

LD LDH LDW LDB

Note Notes point out important information.

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

3.2 Key Features of the ARCv3 ISA

The ARCv3 ISA introduces a number of changes and enhancements, compared to ARCv2, that are common to both 32-bit and 64-bit versions of ARCv3.

- New floating-point instruction set, including a separate floating-point register file, and support for half-precision floating-point values
- Relaxed memory ordering, supporting acquire/release semantics

- ARC64 ISA supports signed/unsigned LIMM operands
- Support for C11/C++11 atomic memory operations (AMOs)
- Revised set of operand options for the memory barrier instruction (DMB)
- Revised function prologue/epilogue operations (ENTER/LEAVE), with support for FP registers
- WAIT instruction, merges functionality of WEVT and WLFC

3.2.1 Programming Model

The programmer's model is common to all ARCv3-based processors and allows upward compatibility of code for software that has been compiled for similarly-configured ARCv3 processors.

3.2.2 Core Register Set

ARCv3-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-r26) 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 115.

Additional banks of registers may be configured for fast interrupt response and context switching. See "Banked Interrupt Registers" on page 186.

3.2.3 Instruction Set Compatibility

3.2.3.1 ISA compatibility between ARC32 and ARC64

When an ARCv3 compliant processor is configured to support ARC64, in general it will not be 100% instruction-set compatible with previous ARCv2 cores nor with ARCv3 cores configured to support ARC32.

However, there is a well-defined subset of the ARCv3 ISA that is instruction-set compatible across all configurations of ARCv3 cores. This subset comprises only instructions that appear in the Baseline Profile for ARC32 (that is, excluding the use formats that are defined in the Compact, Vector DSP, and ARC64 Profiles as well as optional instructions and registers such as those configured with the DSP option), provided the following additional constraints are met:

- All memory addresses computed and used by the program are in the lower 4GB of memory,
- Software does not rely on the layout of data pushed onto the stack during interrupt entry,
- Memory mapping and memory protection mechanisms are not enabled, and

Such baseline compatible binaries could run successfully on any 32-bit or 64-bit ARCv3 core, regardless of configuration. A binary that conforms to the above constraints, and which uses only those instructions from the ARCv3 Compatibility Subset, would run successfully on any ARC32, or ARC64 core.

Such universal binaries could be used, for example, to examine BCRs and check the configuration of a core and boot the system software accordingly.

3.2.4 Auxiliary Register Set

The auxiliary register set contains status and control registers, which by default are 32 bits wide. These auxiliary registers occupy a separate 32-bit address space from the normal memory-access (i.e. load and

store) instructions. Auxiliary registers accessed using distinct Load Register (LR), Store Register (SR), and Auxiliary EXchange (AEX) instructions. See "Auxiliary Register Set" on page 124. The ARC64 programming model extends the default width of each auxiliary register to 64 bits, and provides 64-bit versions of the Load Register Long (LRL), Store Register Long (SRL) and Auxiliary EXchange Long (AEXL) instructions.

3.2.5 32-bit Instruction Formats

The majority of instructions in the ARCv3 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. Dynamic 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.

3.2.6 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 (r30), FP (r27), SP (r28)), and PC (see Program Counter, PC).
- A reduced number of distinct operands, limited to 1 or 2 operand registers, by sharing the destination register with one of the source registers
- Reduced immediate data sizes
- Reduced branch range from maximum offset of ±16MB to maximum offset of ±512B
- No branch delay slot execution modes
- No conditional execution
- No flag setting option (only few instructions set flags, for example, BTST_S (see BTST), CMP_S (see CMP), and TST_S (see TST))

3.2.7 64-bit Data Models and Operating Modes

- The ARCv3 architecture supports both 64-bit and 32-bit data models. It provides a full instruction set supporting operations on data types up to 64 bits wide, and a subset of 32-bit instructions that is compatible with ARCv3 processors. ARC64 supports LP64 and LLP64 data models

3.2.8 Operating Privileges

The ARCv3 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 ARCv3 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

3.3 Configurability

The ARCv3 architecture offers several well-defined methods for configuring the programming model and instruction set. In addition, each processor within the ARCv3 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 ARCv3 implementation, and are therefore described in the relevant *Databook* for each processor.

Table 3-1 describes the configuration options for the programming model supported by the ARCv3 architecture.

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

3.3.1 Programming Model Options

Various aspects of the programming model of an ARCv3 processor can be configured. These include the byte ordering of memory, the number of implemented bits in each address or program counter value, the configuration of the general-purpose register file, and the configuration of the interrupt mechanism. The full set of programming model options is listed in Table 3-1.

Table 3-1 ARCv3 Programming Model Configuration Options

Configuration Option	Value Range	Description
`-rgf_num_regs`	32	Defines the number of registers in the primary register bank supported by the processor. If 16 registers are selected, only registers in the range R0-R3, R10-R15, and R26-R31 are present.
`-rgf_banked_regs`	32	Defines the number of registers replicated per register bank.
`-rgf_num_banks`	1, 2	Defines the number of register file banks. Register bank 0 to Register bank 1.
`-intvbase_ext`	false, true	Specifies if the reset value of the INT_VECTOR_BASE register is supplied by the external input bus intvbase_in. If true, this option overrides -intvbase_preset.
`-intvbase_preset`	PC range	(INTVBASE_PRESET x 1024) Defines the reset value of the programmable INT_VECTOR_BASE register. This parameter is also reflected in the ADDR field of the read only VECBASE_AC_BUILD register. On reset, program execution begins at the address referenced by the reset vector fetched from the 1 KB aligned address (INTVBASE_PRESET x 2048).
`-has_interrupts`	false, true	Indicates if the processor configuration includes the interrupt unit. false: indicates that the interrupt unit is not included. true: indicates that the interrupt unit is included.
`-halt_on_reset`	false, true	Indicates whether the core is halted initially on reset. false: core is not halted initially on reset. true: core is halted initially on reset.
`-number_of_interrupts`	1 to 240 interrupts	Defines the number of interrupts in the interrupt controller. Note: This option is available only if -has_interrupt==true.
`-number_of_levels`	1 to 16	Defines the number of interrupt priority levels (0 to 15) in the interrupt controller. Note: This option is available only if -has_interrupt==true.
`-firq_option`	false, true	Indicates whether the fast interrupt option is enabled. false: indicates that the fast interrupt option is disabled. true: indicates that the fast interrupt option is enabled. Note: This option is available only if `-has_interrupt==true`.

3.3.2 Processor Component Options

ARCv3 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 *ARCv3-based processor databook*.

3.3.3 ISA Options

Table 3-2 summarizes the set of supported instruction set options.

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".

Table 3-2 Instruction Set Options

ISA Extension Pack	Value	Additional Instructions	Additional Aux Registers
`Baseline`		XBFU, XBFUL NORM, NORMH, FFS, FLS, NORML, FFSL, FLSL PREALLOC, SWI_S n6, DSYNC, DMB SWAP, SWAPE, LSL16, LSR16, SWAPL, SWAPEL ASR16, ASR8, LSR8, LSL8, ROL8, ROR8 Multi-bit shift or rotate operations: ASL, LSR, ASR, ROR, ASL_S, LSR_S, ASR_S (excluding b,b,u5 operand format), ASLL, LSRL, ASRL MPYW, MPYUW, MPYW_S, MPYUW_S, MPY, MPY_S, MPYM, MPYMU, MPYU, MAC, MACU, DMPYH, DMPYHU, DMACH, DMACHU, VADD2H, VSUB2H, VADDSUB2H, VSUBADD2H, MPYD, MPYDU, MACD, MACDU, VMPY2H, VMPY2HU, VMAC2H, VMAC2HU, QMPYH, QMPYHU, DMPYWH, DMPYWHU, QMACH, QMACHU, DMACWH, DMACWHU, VADD4H, VSUB4H, VADDSUB4H, VSUBADD4H, VADD2, VSUB2, VADDSUB, VSUBADD, VMAX2, VMIN2	
`-atomic_option`	0	Baseline atomic memory operations are supported, that is, EX, EX.DI, and EXL	-
	1	As option 0, plus the load-locked, store-conditional, and related operations are supported. This set comprises: LLOCK, LLOCK.DI, SCOND, SCOND.DI, WLFC, WAIT, LLOCK, SCOND, LLOCKL, SCONDL, WLFC, WAIT (requires `-ll64_option` true),	
	2	As option 1, plus the ATLD atomic memory instructions.	
	2	ATLDL	
	3	As option 2, plus the ld/st memory ordering options <.aq> and <.rl> are supported.	

Table 3-2 Instruction Set Options (Continued)

ISA Extension Pack	Value	Additional Instructions	Additional Aux Registers
`-code_density_option`[a]	0	-	-
	1	SETEQ, SETNE, SETLT, SETGE, SETLO, SETHS, SETLE, SETGT, SETEQL, SETNEL, SETLTL, SETGEL, SETLOL, SETHSL, SETLEL, SETGTL, ENTER_S, LEAVE_S, BI, BIH, LD_S.AS a,[b,c], LD_S R0-1,[GP,s11]	
	2	As code density option 1, and: LD_S R0-3,[h,u5] LD_S.AS a,[b,c] LD_S R1,[GP,s11] ST_S R0,[GP,s11]	-
	3	As code density option 1, and: JLI_S	JLI_BASE
`-div64`	false	-	-
	true	Indicates if 64/64 divide and remainder instructions are included. The instructions supported by this option are: DIVL, DIVUL, REML, REMUL	
`-fp_dp_option`	false	-	-
	true	Double-precision ARCv3 floating-point instructions	-
`-fp_div_option`	false	-	-
	true	Single-precision ARCv3 floating-point division and square-root instructions, plus floating-point division instructions for data types defined by inclusion of half-precision and/or double-precision instructions.	-
`-fp_hp_option`	false	-	
	true	Half-precision ARCv3 floating-point instructions	-
`-fp_num_regs`	8, 16, 32	-	-
`-fp_vec_option`	false	-	-
	true	Single-precision vector floating-point instructions, plus vector floating-point instructions defined by inclusion of half-precision and/or double-precision instructions	VFPU_STATUS

Table 3-2 Instruction Set Options (Continued)

ISA Extension Pack	Value	Additional Instructions	Additional Aux Registers
`-fp_wide_option`	false	-	-
	true	Double-width versions of vector floating-point instructions defined by -fp_vec_option, and the -fp_dp_option and -fp_hp_option	-
`-has_fp`	false	-	-
	true	Single-precision ARCv3 floating-point instructions	FP_CTRL, FPU_STATUS
`-mpy64`	false	-	-
	true	Indicates if 64x64 multiply instructions are included. The instructions supported by this option are: MPYL, MPYMUL, MPYML, MPYMSUL	
`-m128_option`	false	-	-
	true	Indicates whether 128-bit load and store operations are supported in the core. Replacement of -ll64_option in ARC64.	
`-mpy64_impl`	fast, slow	Selects between full hardware support for 64x64 multiply operations (fast) or a slower hardware emulation of 64x64 multiply operations. The actual cycle count is implementation dependent.	
`-shift_option`	3	All instructions included by `-shift_option` values 1 and 2.	-

a. Some implementations may not distinguish between intermediate levels of code density option, offering the entire set of code density options a single "all or nothing" option. See your processor's Databook for further information on the interpretation of code density options on your ARC processor.A vector unit optionally configured with an ARC HS core displaces some instructions associated with the code density options. See the ISA_PROFILE register for information that indicates how code density related op-code groups are utilized in a given configuration.

3.4 Custom Extensions

The ARCv3-based processor is designed to be extend able 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 ARCv3-based processor.

3.4.1 Extension Core Registers

The core register set has a total of 64 different addressable registers. Registers r32 to r57 are available for extension purposes. Figure 6-2 shows the core register map.

3.4.2 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 base-case 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 ARCv3-based hardware. The Build Configuration Registers contain the version of each ARCv3-based extension and also the build-specific configuration information.

Some optional components in an ARCv3-based processor system may provide only version information registers to indicate the presence of a given component. These *version registers* are not necessarily part of the Build Configuration Registers set. These optional component version registers may be provided as part of the extension auxiliary register set for a component.

3.4.3 Extension Instructions

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

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

- 1 reserved extension group
- 1 user extension group

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

3.4.4 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 ARCv3-based processor which use only the internal condition flags from the status register (Z, N, C, V). For more information, see Table 8-10.

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

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

4 Data Organization and Addressing

This chapter describes the data organization and addressing used by the ARCv3-based processor.

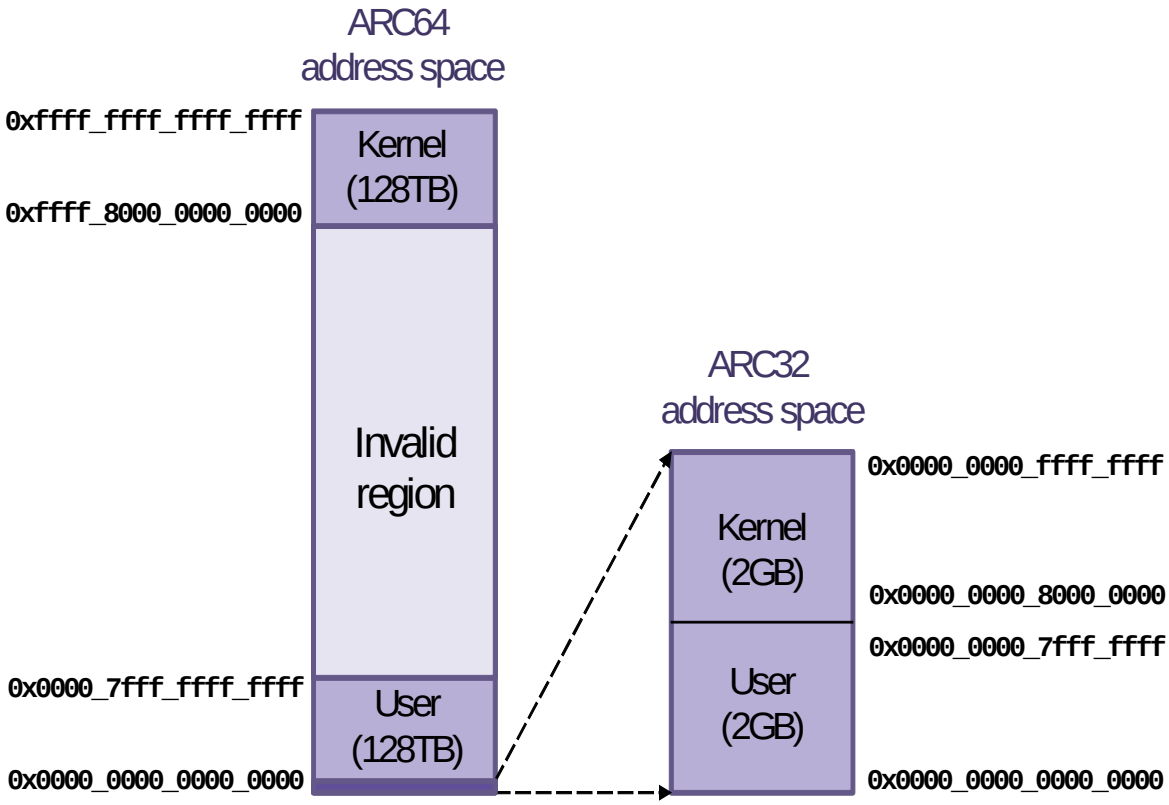
4.1 The ARCv3 Address Space

An ARCv3 processor supports the following three distinct address spaces:

- A half-word addressable Instruction Address Space
- A byte-addressable Data Address Space.
- A 64-bit auxiliary address space. This address space provides an additional 4G word locations that are typically used to access the Control and Status registers, I/O devices, Build Configuration registers, and Application-specific Customer Extension registers. This space is word-addressable using LR, SR and AEX instructions. See Figure 4-1 for more information. All ARCv3-based processors have physically independent Instruction and Data paths that allow for Von Neumann or Harvard configurations. However, the default memory configuration for the processor unifies the Data and Instruction memory spaces. A load or store to the memory address location *nn* in the data memory accesses location *nn* in the instruction memory.

4.2 Operations on 64-bit Addresses

Figure 4-1 Unified Address Spaces in ARC64 and ARC32 modes



All instructions that expect address operands, or compute addresses as their results, operate at 64-bit precision for example: load and store operations, where addresses are always 64-bit in ARC64. Instructions that deal with program address such as branch-and-link instruction (BLINK) are also performed at 64-bit precision.

ARC64 core products define the actual number of virtual and physical address bits they implement. If the number of checked virtual address bits is V, and V < 64, then virtual address bits [63:V] are assumed to be the same as bit V-1, and are not checked. If the number of implemented physical address bits P is < 64, then virtual memory management schemes do not map bits [63:P], and physical memory protection/checking schemes ignore address bits [63:P].

4.3 Load and Store Instructions

The ARCv3 ISA introduces some new load and store instructions to support the following capabilities:

- Acquire/release semantics for release consistency and relaxed memory ordering
- 64-bit load/store operations in ARC64
- Optional 128-bit load/store operations in ARC64

The new load/store instructions are encoded without altering the encoding of pre-existing load/store instructions.

4.3.1 64-bit Sign-extension for LDB.X and LDH.X

In ARC64, the LDB.X and LDH.X instructions sign-extend their result from the most significant bit of the byte or half-word of load data all the way to bit 63 of the destination register.

4.4 Data Formats

All ARCv3-based processors support little-endian byte ordering by default.

The processor can operate on data of various sizes. The memory operations (load and store type operations) can have data of 64 bits (long-word) , 32 bits (Word), 16 bits (Half-word), or 8 bits (Byte).

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.

Long-word (64-bit) operations are supported only in ARC64 and use a single 64-bit register as the source of store data and the destination of load data.

4.4.1 64-bit Data Processing Instructions

In addition to the mandatory instructions to handle 64-bit addresses (load, store, address adjustments, branches, and so on), ARC64 also includes instructions for 64-bit data processing. That is, common operations on datatype 'long int' can be executed by a single instruction.

4.4.2 32-Bit Data Operations

ARCv3 ISA includes explicit instructions that operate on 32-bit data set so that the compiler can indicate to the hardware which operations require no more than 32-bits of precision. When operating on 32-bit data, the upper 32 bits of their destination register are set to zero. The upper 32 bits of any source operand are ignored.

4.4.3 Register Usage in 32-bit Instructions

When a 64-bit core register is accessed as a 32-bit source operand, the value is obtained from the lower 32 bits of the operand register, and the upper 32 bits are unused.

When a 64-bit core register is assigned a 32-bit result, the result bits are assigned to the lower 32 bits of the register and the upper 32 bits of the register are cleared.

4.4.4 Data Model for Size int

Even with a fairly complete set of 64-bit data operations included in the ARC64 ISA, the ARC64 ABI specifies that the C int datatype corresponds to a 32-bit two's complement word. To access a 64-bit integer, the programmer must either use a `long int` (LP64 data model) or a `long long int` (LLP64 data model). The advantages of the LP64 and LLP64 data models over ILP64 (where sizeof(int) == 8 bytes) include:

- Memory efficiency: no space wasted when applications do not need 64-bit integers

- Software portability: complying with a defacto industry standard reduces the effort to port a program to ARC. The common programming error of equating sizeof(int) to sizeof(void*) is easily flagged by a compiler.

4.4.5 Vector Operands

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

4.4.5.1 64-Bit Operand Data Elements

In ARC64, the general purpose registers are 64 bits wide, allowing each register operand to supply a full 64-bit operand.

4.4.5.2 LDDL and STDL Operands

The Load Double (LDDL) and Store Double (STDL) instructions are provided when M128_OPTION is enabled. These instructions load (or store) a pair of 64-bit registers from (or to) memory. Effectively these instructions move a vector of two 64-bit objects between a pair of registers and memory. The ARC 64-bit ABI stores 128-bit objects in memory as a vector of two 64-bit values, and hence LDDL and STDL can be used to access 128-bit objects that are laid out in memory according to the ARC 64-bit ABI.

When LDDL and STDL instructions are executed, the low word address always goes with the even register Rn and the high word address (low+8) goes with the odd register Rn+1.

Consider the following double load instruction:

```
LDDL Rn,[A]
Rn+1 ← { A+15, A+14, A+13, A+12, A+11, A+10, A+9, A+8 }
Rn   ← { A+7, A+6, A+5, A+4, A+3, A+2, A+1, A }
```

The values in R_n and R_{n+1} are the same as they would be if the same 128-bit value had been passed as a function argument using two consecutive registers, Rn and Rn+1. The 64-bit word at address A is passed through Rn, and the 64-bit word at address A+8 is passed through Rn+1.

4.4.6 32-Bit Operand Data Elements

Figure 4-2 32-Bit Elements in a 32-bit Operand

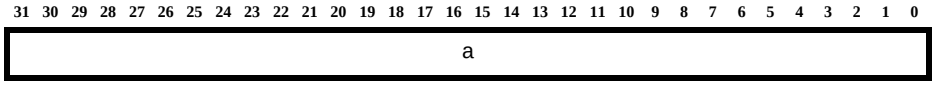
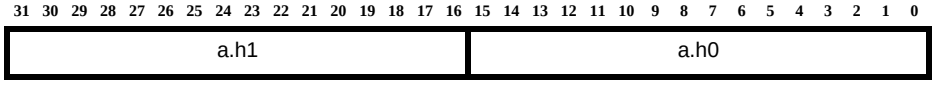


Figure 4-3 16-Bit Elements in a 32-bit Operand



4.4.7 Expansion of Long Immediate Literals

When a LIMM is required as a scalar operand in a 64-bit context, the ARC64 ISA provides two variants of the LIMM; one that zero-extends the LIMM to 64 bits, and another that sign-extends the LIMM to 64 bits.

The zero-extended version is specified using the existing LIMM indicator of r62. The new sign-extended version is specified using a new LIMM indicator of r60.

The assembler provides syntax to specify signed and unsigned 32-bit constant integers, allowing software to tell the assembler whether to use r62 or r60 as the LIMM indicator.

Figure 4-4 Zero Extension of a LIMM in r62

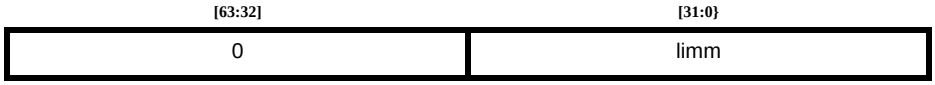
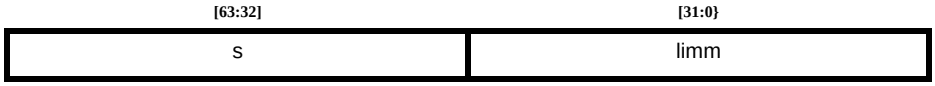


Figure 4-5 Sign Extension of a LIMM in r60



4.4.8 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.

When a long-immediate operand is supplied as the store data to an STDL instruction, the LIMM is extended to 64 bits according to the normal rules (zero-extended for r62, sign-extended for r60), and then duplicated to form a 2-element vector of identical 64-bit immediate values.

Figure 4-6 and Figure 4-7 illustrate expansion of a shimm to a word (32 bits).

Figure 4-6 Expansion of a u6 Literal to a 32-Bit Word (w-shimm)

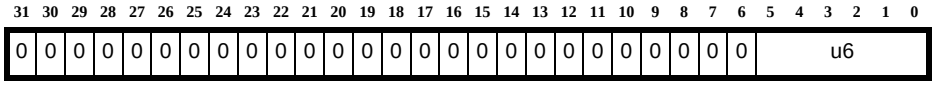


Figure 4-7 Expansion of an s12 Literal to a 32-Bit Word (w-shimm)

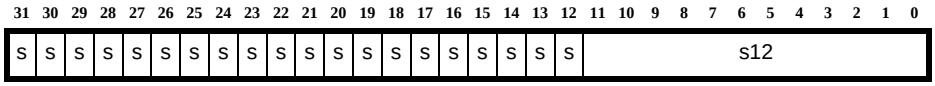


Figure 4-8 and Figure 4-9 illustrate expansion of a shimm to a half-word (16-bits).

Figure 4-8 Expansion of a u6 Literal to Half-Word (16-bit, hw-shimm)

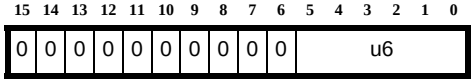
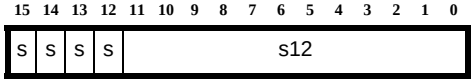


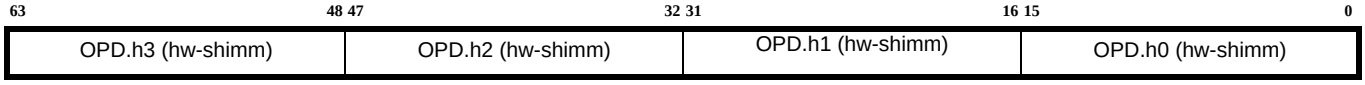
Figure 4-9 Expansion of an s12 Literal to Half-Word (16-bit, hw-shimm)



4.4.9 Expansion of Literals in Vector Instructions

When a vector instruction expects 16-bit elements as a source operand, a shimm operand is replicated to the required operand size, and a limm operand (containing two 16-bit values) is duplicated if a 64-bit operand is required as shown below:

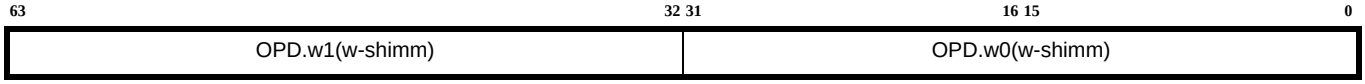
Figure 4-10 A Half-Word (16-bit) Short Limm Value Replicated to form a 64-Bit Operand



Where a hw-shimm represents u6 or s12 extended to half-word (16-bits). For an illustration of hw-shimm, see Figure 4-8 and Figure 4-9. When a limm operand containing h1 and h0 is duplicated to create a 64-bit operand, h3 will be a copy of h1 and h2 will be a copy of h0.

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

Figure 4-11 A word (32-bit) Short-limm or Limm value Replicated to form a 64-Bit Operand



Where a w-shimm represents u6 or s12 expanded to a word (32-bits), or a limm. For an illustration of w-shimm, see Figure 4-6 and Figure 4-7.

Figure 4-12 and Figure 4-13 illustrate the expansion of two hw-shimm(16 bits) to a word (32 bits).

Figure 4-12 Two 16-Bit Vectors with Expanded u6 Operands

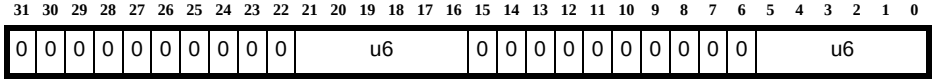


Figure 4-13 Two 16-Bit Vectors with Expanded s12 Operands

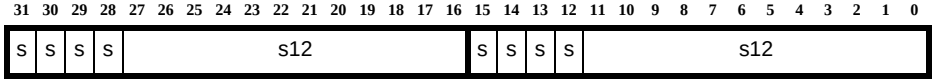


Figure 4-14 and Figure 4-15 illustrate the expansion of four 8-bits vectors (expanded u6 or s12 literals) to a word (32 bits).

Figure 4-14 Four 8-Bit Vectors with Expanded u6 Operands

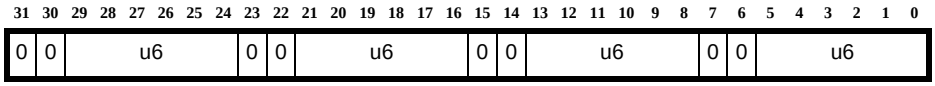
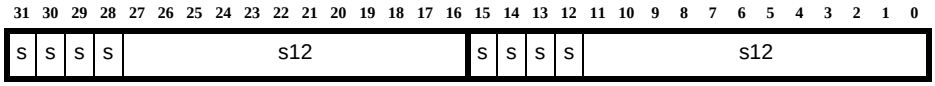


Figure 4-15 Four 16-Bit Vectors with Expanded s12 Operands



4.4.10 Expansion of Literals in Store Double (STD)

When a long-immediate operand is supplied as the store data to an STDL instruction, the LIMM is extended to 64 bits according to the normal rules (zero-extended for r62, sign-extended for r60), and then duplicated to form a 2-element vector of identical 64-bit immediate values.

4.5 Data Layout in Memory

Baseline ARCv3 architectures access data memory using byte addresses, with little-endian ordering of bytes within multi-byte accesses, and require that all memory addresses are aligned as follows:

- 128-bit Double long-words are aligned to 32-bit word boundaries (requiresM128_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 ARCv3 architecture, all misaligned data accesses raise a data misalignment exception. Certain high performance versions of the ARCv3 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.

4.5.1 128-Bit Data

Figure 4-16 shows the little-endian representation in byte-wide memory.

Figure 4-16 128-bit Register Data in Byte-Wide Memory, Little-Endian

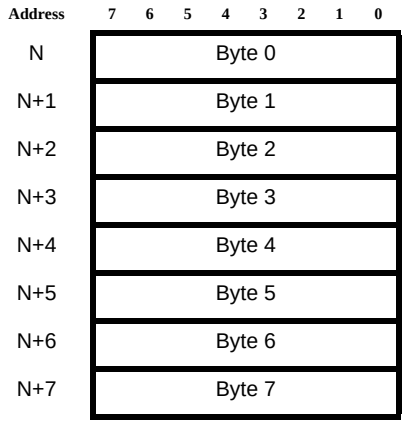
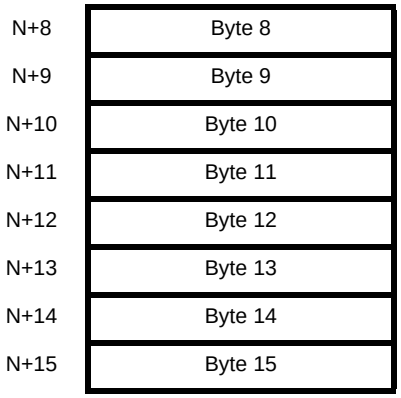


Figure 4-16 128-bit Register Data in Byte-Wide Memory, Little-Endian (Continued)



For information about the 64-bit data organization in a register pair, see Figure 4-2.

4.5.2 32-Bit Data

All load or store, arithmetic, and logical operations support 32-bit data. Figure 4-17 on page 104 show the data representation in a general purpose register. Figure 4-18 on page 104 shows the little-endian representation in byte-wide memory.

Figure 4-17 Register containing 32-bit Data

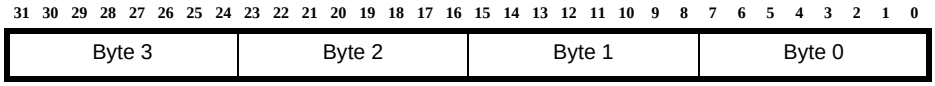
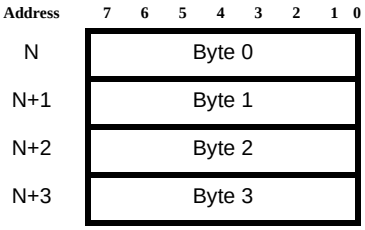


Figure 4-18 32-bit Register Data in Byte-Wide Memory, Little-Endian



4.5.3 16-Bit Data

Load, store, and some multiplication instructions support 16-bit data. 16-bit half-word data can be promoted explicitly to an equivalent 32-bit word value by using unsigned extend (EXTH (see EXTH)) or

signed extend (SEXH (see SEXHL)) instructions. Figure 4-19 shows the 16-bit data representation in a general purpose register.

For the programmer's model, the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a *data invariance* principle. Figure 4-20 shows the little-endian representation of 16-bit data in byte-wide memory.

Figure 4-19 Register Containing 16-Bit Data

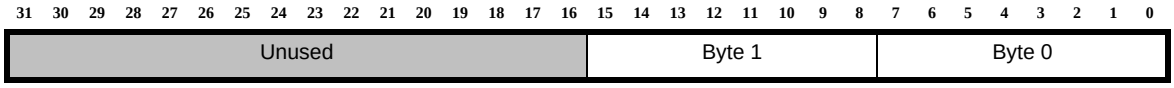
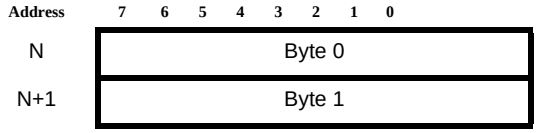


Figure 4-20 16-bit Register Data in Byte-Wide Memory, Little-Endian



4.5.4 8-Bit Data

The load and store operations support 8-bit data which can be promoted to a 32-bit value by using unsigned extend (EXTB) or signed extend (SEXB) instructions. Figure 4-21 on page 105 shows the 8-bit data representation in a general purpose register.

For the programmer's model, the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a *data invariance* principle. Figure 4-22 on page 105 shows the representation of 8-bit data in byte-wide memory.

Regardless of the endianness of the ARCv3-based system, the byte-aligned address, N, of the byte is explicitly given and the byte is stored or read from that explicit address.

Figure 4-21 Register Containing 8-Bit Data

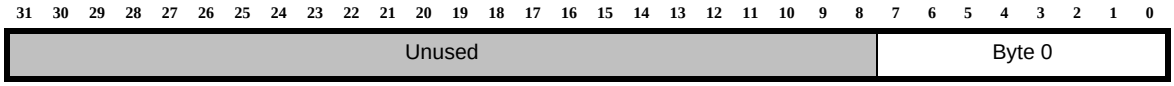
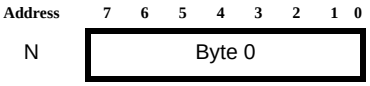


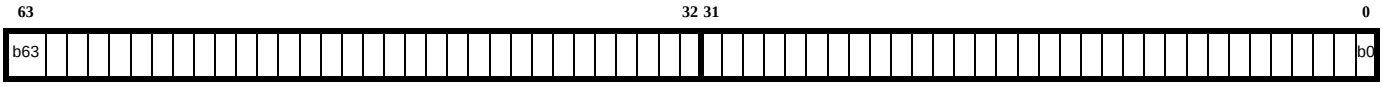
Figure 4-22 8-bit Register Data in Byte-Wide Memory



4.5.5 1-Bit Data

The ARCv3 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, 32-bit or 64-bit data located within core registers because each bit is individually addressable.

Figure 4-23 Register Containing 1-Bit Data



4.6 Instruction Layout in Memory

The ARCv3 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. A32-bit long-immediate data value is similarly represented as a sequence of two half-word values. A 16-bit instruction is encoded as a single half-word.

Figure 4-24 Half-word Numbering of a 32-Bit Instruction or Long-immediate Data Item

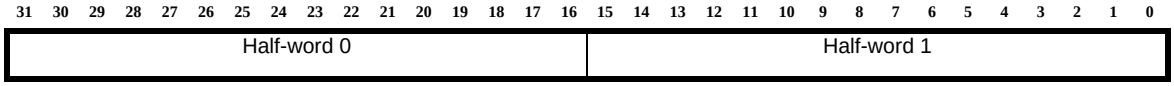


Figure 4-24 shows the number of two half-words in a 32-bit instruction or long-immediate data item.

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 ARCv3-based processor can be configured to support either big-endian or little-endian byte ordering. This configuration is always static, and cannot be altered at runtime.

Figure 4-25 illustrates the little-endian memory address offset of each byte, within a 32-bit instruction or long-immediate value, relative to the start address of that instruction or long-immediate value. Figure 4-26 shows the corresponding big-endian addresses.

Figure 4-25 Little-endian Offset of each Byte in a 32-Bit Instruction or Immediate

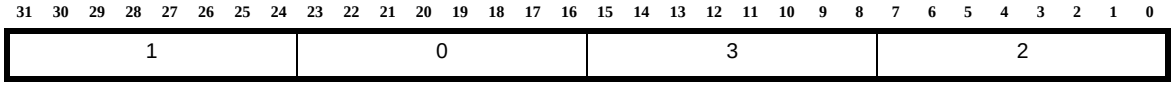
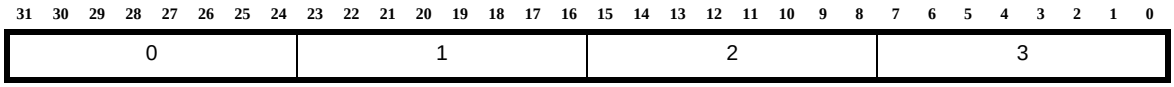


Figure 4-26 Big-endian Offset of each Byte in a 32-Bit Instruction or Immediate



Similarly, Figure 4-27 illustrates the little-endian memory address offset of each byte within a 16-bit instruction, relative to the start address of that instruction. Figure 4-28 illustrates the big-endian memory offset of each byte in a 16-bit instruction.

4.7 Addressing Modes

Figure 4-27 Little-endian Memory Offset of each Byte in a 16-Bit Instruction

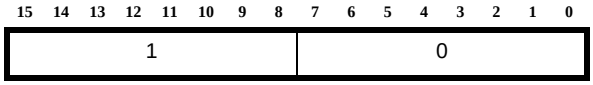
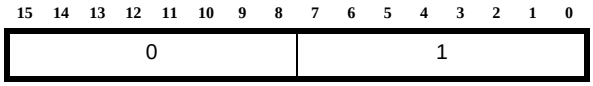


Figure 4-28 Big-endian Memory Offset of each Byte in a 16-bit Instruction



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

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

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

Table 4-1 Instruction Variants

Operation	Description
<.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 (some load and store operations)
<.aa>	address writeback
<.T>	invert the default static branch prediction (see "Branch-and-link Format [F32_BR1, BL]" on page 284)

Table 4-2 Operand Description

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

4.8 Null Instruction Format

The ARCv3 ISA supports a special type of instruction format, where the destination of the operation is defined as null (0). When this instruction format is used, the result of the operation is discarded, but the status flags may be set depending on <.f> or the instruction. This design allows any instruction to act in a manner similar to compare.

Example 4-1 Null Instruction Format

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

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

4.9 Conditional Execution

A number of the 32-bit instructions in the ARCv3 ISA support conditional execution. A 5-bit condition code field allows up to 32 independent conditions to be tested before execution of the instruction.By default, 16 conditions are defined with the remainder available for customer definition, as required.

4.10 Conditional Branch Instruction

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

4.11 Compare and Branch Instruction

The ARCv3 ISA includes two forms of instruction that combine a comparison and branch.

The Compare and Branch Conditionally (BRcc) instruction combine a compare instruction (CMP) with a conditional branch (Bcc) instruction. These instructions are available in both 32-bit and limited 16-bit encodings.

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 in 32-bit encodings.

5 ARCv3 Memory Consistency Model

This section defines the memory consistency model for the ARCv3 ISA.

A memory model defines the values that are returned when reading memory by defining the order in which writes to memory can be observed by each read from memory. The ordering of memory operations is formally specified as a set of rules that must be obeyed by hardware. In general, relaxed memory ordering rules allow greater freedom for hardware to optimize memory performance. However, hardware is not obliged to exploit the full extent of relaxed orderings permitted by the memory model; each implementation is free to implement memory orderings that are stricter than actually required by the rules. Memory ordering rules should therefore be interpreted as placing limits on the extent to which memory operations can be unordered, when viewed globally, compared to the order in which they are specified in a program.

5.1 Terminology for Memory Ordering Rules

A thread is the execution of a sequence of instructions obtained by fetching instructions from successive values of the program counter.

A program is shorthand for a shared-memory multi-threaded program. This is a collection of threads, and information may be communicated between threads by reading and writing to common memory locations.

Program Order defines the order in which two operations occur in the sequential execution of a thread. If a thread executes two operations A and B, and A appears before B in sequential execution, then A occurs before B in program order[1].

The ARCv3 ISA supports a range of memory instructions, each of which performs a memory access. These are read accesses, write access, or in some cases both a read and a write access. Each memory access is defined in terms of its address and the size (in bytes) of the data that is accessed. If the address is a multiple of the size, then the access is aligned, otherwise it is non-aligned.

Memory accesses are implemented in hardware using one or more memory operations. A memory operation is a low-level read or write operation to a physical memory. All aligned memory accesses are performed with exactly one memory operation. In contrast, non-aligned memory accesses comprises two or more memory operations. Memory operations arising from the same memory access are not ordered with respect to each other, and they are not obliged to share the same global memory order. However, all memory operations from one memory access share the same program order.A flow dependency is defined as a relationship between two memory operations A and B in the same thread, whereby information

1. In practice this allows hardware to split a non-aligned memory access into a pair of smaller aligned memory operations to adjacent words, for example, if the memory access straddles a cache-line boundary. The memory ordering rules allow a memory operation from a different threads to occur in between the split memory operations, in the global memory order.

produced by A is used to determine B's behavior. There are three types of flow dependency; address, data, and control. An address dependency occurs if the result of A influences the address used by B. A data dependency happens if B is a store, and the result of A is used as the store data for B. A control dependency occurs if any result from A influences a conditional branch/jump instruction that precedes B; these results may include load data or the status result from a store-conditional instruction. Flow dependencies may be direct or indirect. An explicit dependency links A to B through a register. An indirect dependency links A to B through any number of intervening operations, including memory operations. Thus, a flow dependency can be transmitted through registers and memory locations (for example, when a register is spilled and later reloaded).

Concurrency covers both inter-thread and intra-thread concurrency. Inter-thread concurrency means that the instructions of different threads in a program are interleaved in time, and their order is non-deterministic. Intra-thread concurrency implies that the instructions of the same thread may execute in any order provided that:

- they yield result values that are indistinguishable from the sequential execution of those instructions in program order; and
- the order of memory operations within a thread does not violate the memory ordering rules.

Global Memory Order defines a total order over all memory operations in a program. The global memory order is defined by the sequence in which memory operations perform.

- A load operation performs when it determines its load data.
- A store operation performs when its store data becomes globally visible.

An atomic memory operation is either a load instruction with acquire semantics or a store instruction with release semantics. Acquire and release semantics are defined by the memory ordering rules given in "Memory Ordering" on page 112. The memory operations associated with an atomic memory instruction must be performed atomically in memory. For example, it is not permissible to perform an unaligned atomic memory operation as two aligned memory operations if those operations are divisible. This rule ensures that each atomic memory operation occupies a single unique position in the global memory order.

A non-atomic memory operation is a load or store that does not have acquire or release semantics.

A locking memory operation is either a load-locked or a store-conditional instruction.

A memory barrier operation is an instruction that enforces a point of ordering between preceding memory operations and future memory operations. DMB is the instruction that implements memory barriers in ARCv3. DMB is optionally capable of specifying orthogonally the categories of memory operation over which the memory barrier is to be enforced. These categories are:

- **Read:** load or pop instructions, including those from LEAVE_S and interrupt or exception epilogues.
- **Write:** store or push instructions, including those from ENTER_S and interrupt or exception prologues.
- **Control:** cache or TLB maintenance operations, and all SYNC, DSYNC, and DMB instructions.

5.2 Memory Ordering

The ARCv3 ISA supports a weak ordering of memory operations defined by a number of axioms. In these axiomatic definitions, A and B are any two memory operations[2] occurring in the same thread and A always precedes B in program order. *A and B overlap* if A and B access any common bytes in memory.

1. Loads return the values most-recently written to the bytes specified by their address and data size, either in program order or global memory order (whichever is most recent).
2. The global memory order is non-deterministic.
3. The ARCv3 memory model is multi-copy atomic (MCA). In a multi-core implementation this means that all cores observe stores in the same global memory order. As a performance optimization, a thread may observe its own stores early (for example, if they are forwarded to a load from the store buffer), but such forwarded stores must be flushed before completing a store-release if there could be any other shared copy of that value in another core's cache.
4. Hardware must always respect the following rules that define global memory order:
   a. If A and B overlap, and B is a store, then A must precede B in global memory order. Thus, if A is a load, then it must determine its value before store B becomes globally visible. Conversely, if A is a store, then it must become globally visible before store B becomes globally visible.
   b. If A and B overlap, and both are loads, then A must precede B in global memory order unless:
      i. the result for B is forwarded from store S that is between A and B in program order, or
      ii. the results for both A and B are completely determined by the same store.
   c. If B has a flow dependency on A, then A must precede B in global memory order, unless B is a load and its dependency on A is purely a control dependency (that is, loads can be speculative).

This case includes the case where the flow dependency traverses a memory location. This situation can occur when the result of load B is obtained from an intervening store S, where S occurs between A and B in program order, and S has an address or data dependency on A.

   d. If A and B overlap, B is a load, and A is either an atomic or a locking memory operation, then the load result of B cannot be forwarded from A until A has performed.
   e. If A is a load with acquire semantics, B follows A in global memory order.
   f. If B is a store with release semantics, A precedes B in global memory order.

This means B cannot be merged with any preceding memory operation and must perform in memory at a different (and later) position in the global memory order. However, although this rule does not strictly prevent a store after B in program order appearing before B in global memory order, implementations typically prevents any atomic store from combining with any other store operation, and hence that situation would not happen in practice.

   g. If A is a store with release semantics, and B is a load with acquire semantics, A precedes B in global memory order.
   h. If B is any memory operation, and there is a memory barrier M with operands that specify memory category sets g1 and g2, and A occurs before M and B occurs after M in program order, then A must precede B in global memory order if A is an operation of category C and C belongs to g2.

2. *Memory operations* includes all memory access instructions, such as loads and stores, as well as memory system control operations, such as cache flush commands.

i. If A is any memory operation, and there is a memory barrier M with operands that specify memory category sets g1 and g2, and A occurs before M and B occurs after M in program order, then B must succeed A in global memory order if B is an operation of category C and C belongs to g2.

5. Hardware must always observe the following rules when executing locking memory operations:

a. The thread that executes a load-locked operation obtains a reservation on the memory location given by its address, which remains in place until it is relinquished.

b. A store-conditional operation performs if and only if its thread holds a reservation on the memory location given by its address when the store-conditional is executed. Store conditional operations normally return a Boolean result to indicate whether they performed, although this is not strictly required by the memory ordering rules.

c. If A and B have the same address M, and A is a load-locked operation whereas B is a store-conditional, then any store to M occurring after A in global memory order or program order, but before B is performed, will kill the reservation obtained by A and consequently B will not perform in memory when it executes.

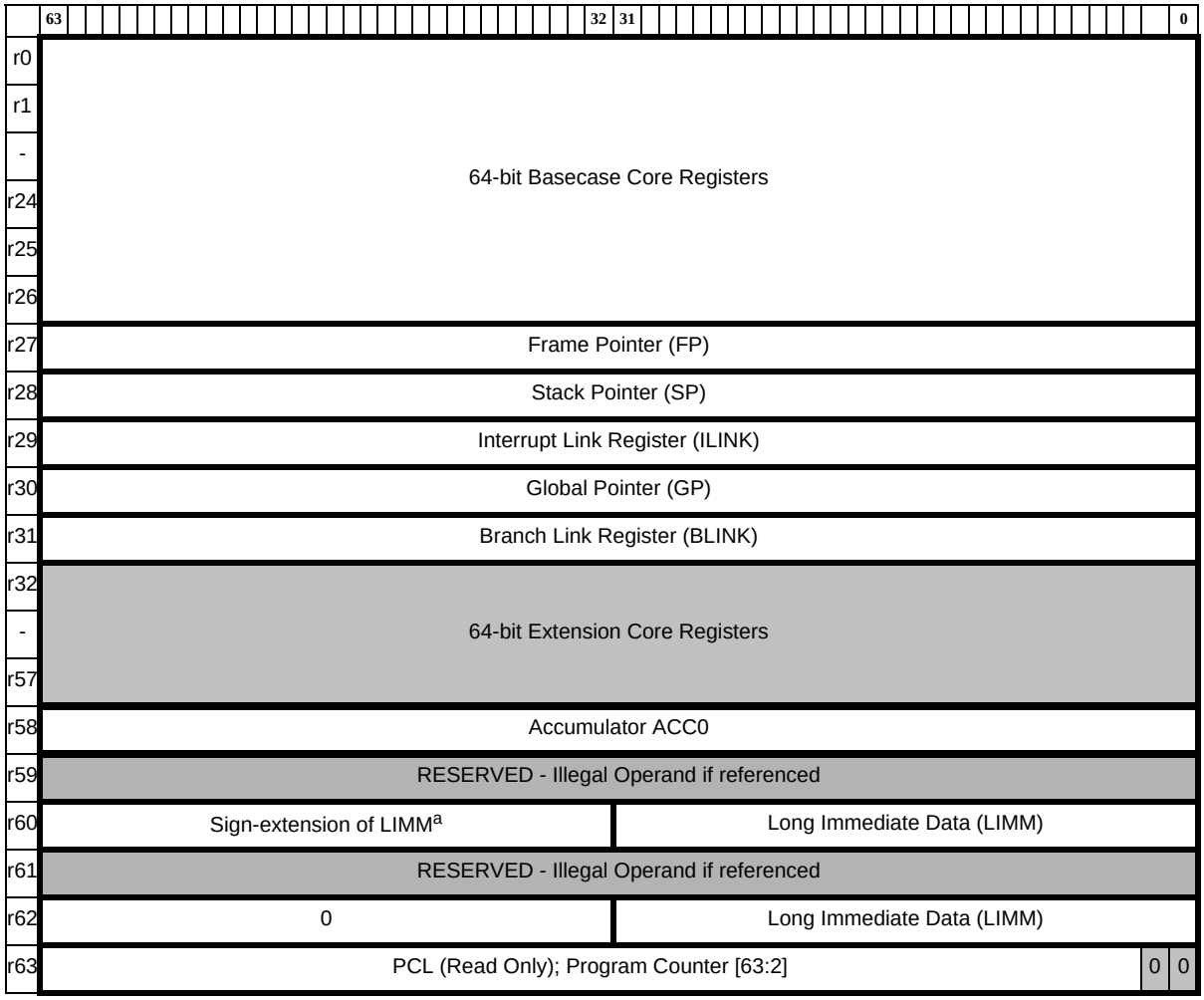
Note As hardware typically maintains one physical memory reservation, in the form of a lock flag and a lock physical address, hardware voluntarily relinquishes the lock if it switches thread.

6 Core Architectural State Registers

6.1 Core Register Set

The ARCv3 programming model provides a set of general-purpose registers. In ARC64, all general-purpose registers are 64 bits wide. Figure 6-1 shows a summary of the core register set.

Figure 6-1 ARC64 Core Register Map Summary



a. In ARC64 there are two alternative LIMM indicator registers, r60 and r62. The r60 register indicates a sign-extended 32-bit immediate, whereas r62 indicates an unsigned 32-bit immediate. These registers can each be used in exactly the same set of circumstances as the ARCv2 LIMM (r62) operand. This includes the use of either r60 or r62 to indicate a null destination.

6.1.1 Core Register Usage in the ABI

The ARCv3 ISA defines the usage of most general purpose registers in common across the 32-bit and 64-bit ARC32 and ARC64 ISAs, and this is reflected in the ARCv3 System V ABI.

A small number of differences remain between ARC64 and ARC32, these are:

- ARC64 does not support zero-overhead loops, and therefore has no need for the LP_COUNT register.
- ARC64 provides two versions of the 32-bit long-immediate data value, one that is zero-extended and one that is sign-extended.
- ARC64 provides a single 64-bit accumulator register.
- ARC64 does not support direct 64bit LIMM.

- For stack frames that do not need a frame-pointer (fp) register, r27 can be added to the set of available callee-saved general-purpose registers. If the function prolog saves all registers r14-r26, then it can also save/restore r27 as a paired load/store with r26. The list of saved/restored registers specified by ENTER_S/LEAVE_S instructions extends to r27 allowing all callee-saved registers (r14 – r27) to be saved using those instructions. It is also possible to save/restore fp independently of the callee-saved register list, as previously defined for ENTER_S/LEAVE_S, in which case it is assumed to contain a stack frame-pointer.
- To minimize the number of GPRs saved and restored by function prologue/epilogue sequences, compilers should allocate callee-saved registers from r14 upwards.

Table 6-1 ARCv3 ARC64 System V ABI Register Usage

Register	Primary Function	16-bit Access	RF16 Available	ENTER_S, LEAVE_S Accessible	Secondary Function
r0	function result; argument 1	Yes	Yes		caller-saved; may be freely used by compiler
r1	argument 2	Yes	Yes		
r2	argument 3	Yes	Yes		
r3	argument 4	Yes	Yes		
r4	argument 5				
r5	argument 6				
r6	argument 7				
r7	argument 8				

Table 6-1 ARCv3 ARC64 System V ABI Register Usage

r8	caller-saved; may be freely used by compiler				-
r9					
r10			Yes		
r11			Yes		
r12		Yes	Yes		
r13		Yes	Yes		
r14	general-purpose registers	Yes	Yes	Yes	callee-saved; may be freely used by compiler
r15		Yes	Yes	Yes	
r16			Yes	Yes	
r17				RF32 only	
r18				RF32 only	
r19				RF32 only	
r20				RF32 only	
r21				RF32 only	
r22				RF32 only	
r23				RF32 only	
r24				RF32 only	
r25				RF32 only	
r26				RF32 only	
r27	frame pointer (fp)		Yes	Yes	callee-saved register variable
r28	stack-top pointer (sp)		Yes	Yes	-
r29	interrupt link register (ilink)		Yes		-
r30	small-data base register (gp)		Yes		-
r31	branch link register (blink)		Yes	Yes	caller saved
r32-r57	customer extension registers (APEX)				caller-saved
r58	64-bit fixed-point accumulator (acc0)				caller-saved register (if configured)

Table 6-1 ARCv3 ARC64 System V ABI Register Usage

r59	Unused - Illegal Operand				
r60	Sign-extended LIMM or null destination operand				caller-saved
r61	reserved – Illegal Operand				-
r62	Zero-extended LIMM or null destination operand.				-
r63	program-counter value (pcl), 4-byte aligned (that is, bits 0 and 1 are always 0)				

6.1.1.1 Return Values

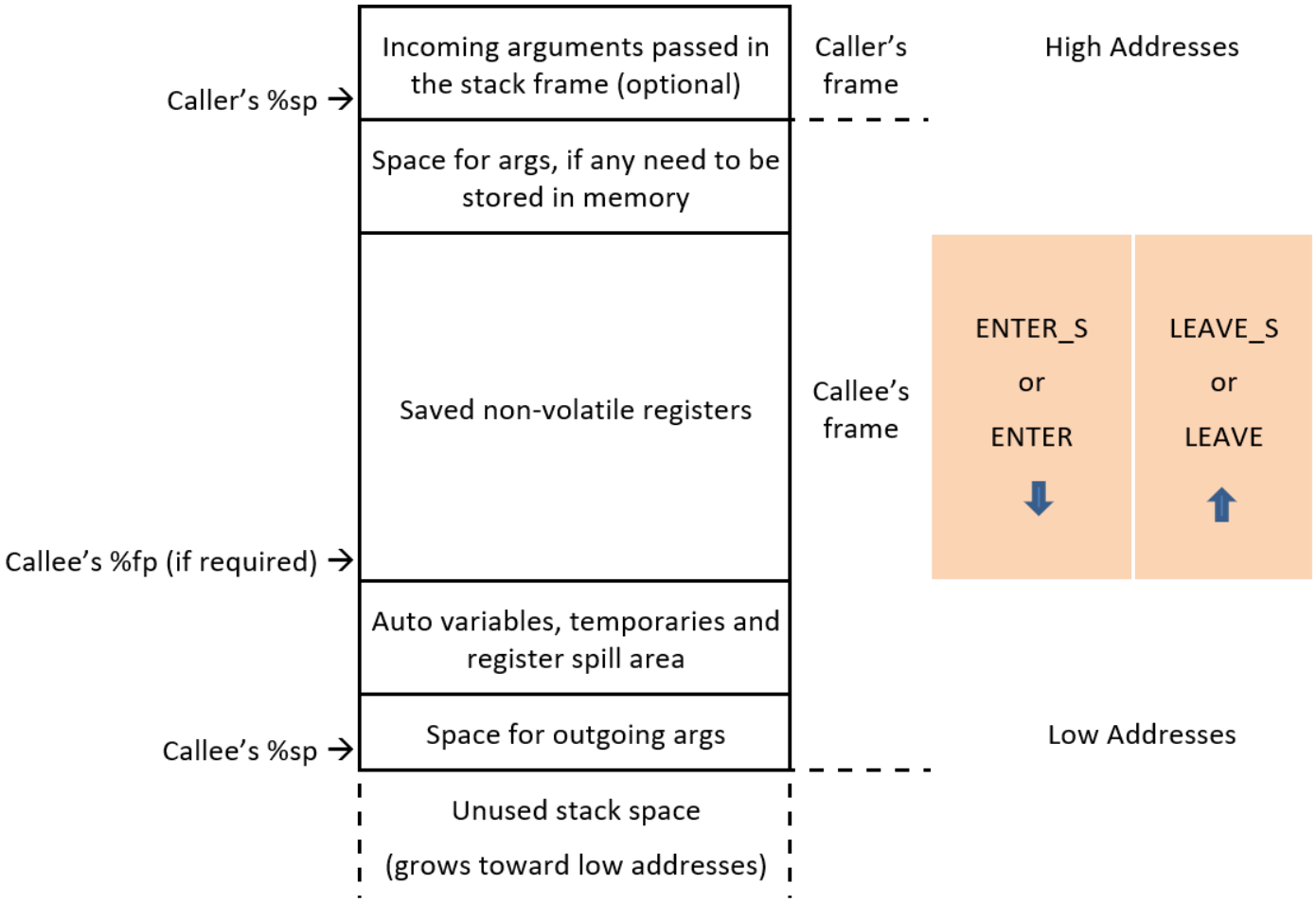
Function results are returned as shown below:

- In ARC64, any fixed-point scalar or pointer type that is 64 bits or less in size (`char`, `short`, `int`, `long`, `long long`) is returned in r0.
- Any scalar floating-point type (`float` and `double`) is returned in f0.
- Results of type `complex float` and `complex double` are returned in f0 and f1.
- Results of type `struct` are returned in a caller-supplied temporary variable whose address is passed in r0. For such functions, the arguments are shifted so that they are passed in r1 and up.

6.1.1.2 Stack Frame Layout in the ARCv3 ABI

Figure 6-2 shows ARCv3 stack frame layout.

Figure 6-2 Stack Frame Layout Defined by the ARCv3 ABI



The ABI does not stipulate the positions for individual registers within the saved non-volatile register region of the stack frame. It is the responsibility of the callee to save these registers during the function prologue and restore them during the function epilogue. The DWARF CFA records tell the debugger all it needs to know about the internals of a stack frame, so this does not need to be specified in any greater detail by the ABI.

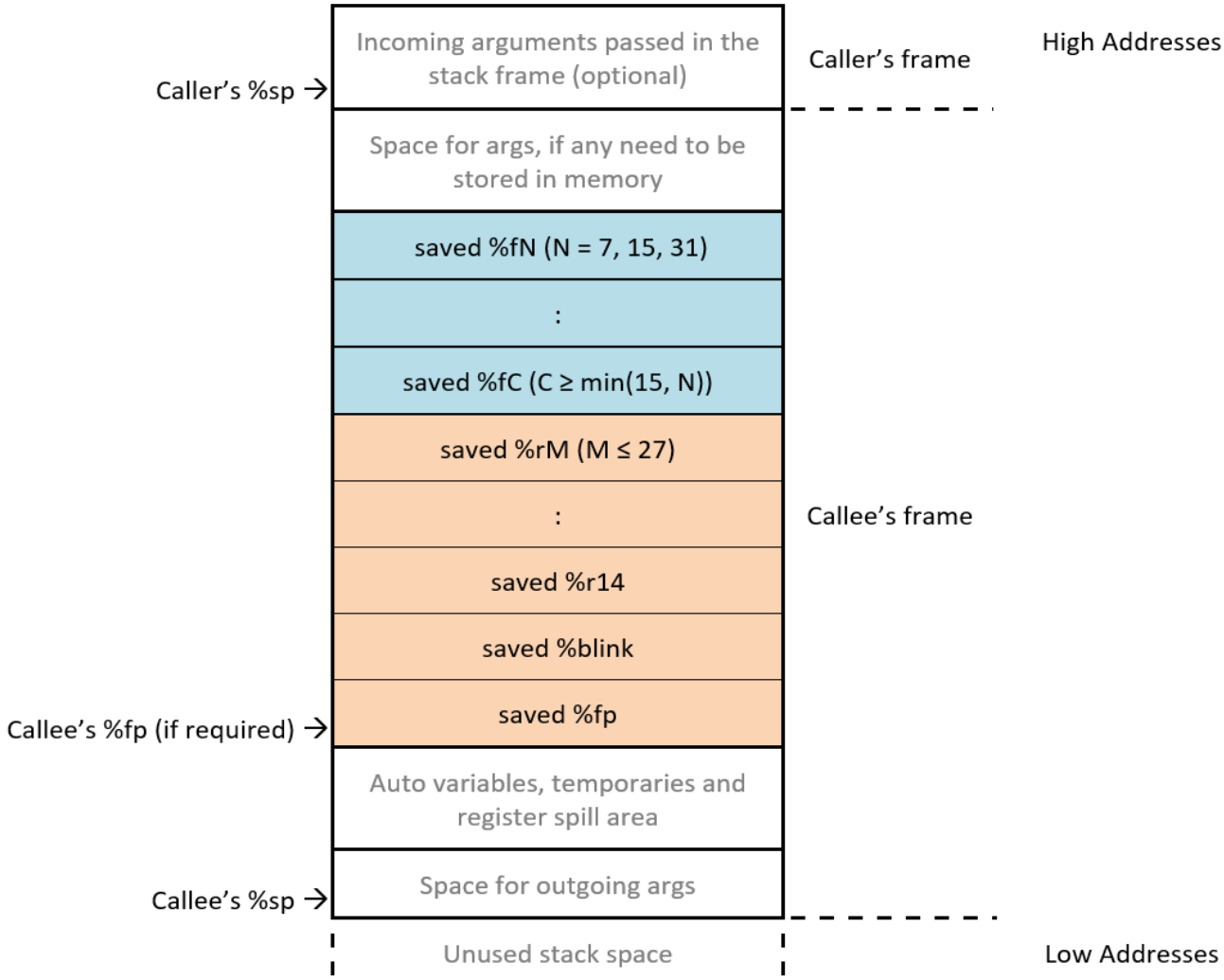
On the right-hand side of the stack frame diagram the colored regions illustrate the regions that can be saved/restored by the compact function prologue/epilogue instructions (ENTER_S, LEAVE_S, ENTER, and LEAVE).

6.1.2 Function Prolog and Epilog Instructions

The ENTER_S and ENTER instructions can be used to save non-volatile registers as part of the function prolog. The LEAVE_S and LEAVE instructions restore non-volatile registers as part of the function epilog. These instructions give each of the callee-saved registers a defined position that they save or restore within the non-volatile region. These instructions provide the compiler the flexibility on what registers to save; the frame pointer and the blink register can be omitted.

Figure 6-3 illustrates the registers' layout within the non-volatile register save area, saved or restored by the ENTER_S, ENTER, LEAVE_S, and LEAVE instructions.

Figure 6-3 Layout of Non-volatile Save Area for ARCv3 Prolog and Epilog Instructions



Compilers are free to save non-volatile registers in different positions, although they are unable to use ENTER_S, ENTER, LEAVE_S and LEAVE.

The ENTER_S instruction saves general-purpose callee-saved registers and provides options to save the special-purpose registers %fp and %blink. The ENTER instruction offers the same capabilities as ENTER_S, but can also save floating-point registers. Conversely, the LEAVE_S instruction restores general-purpose callee-saved registers and any selected special-purpose registers; it may also implement an optional branch back to the return address given by the %blink register. Likewise, the LEAVE instruction has all of the capabilities of the LEAVE_S instruction, with the additional ability to restore floating-point registers.

The callee is responsible for allocating space for auto variables, temporaries, and register spill area, and for outgoing arguments for functions it calls. If the stack frame has a frame pointer (%fp), this local space does

not need to be deallocated by the callee, as this will be done automatically by the LEAVE_S instruction when it sets %sp to %fp.

When `-has_fp` is `true` the actual number of floating-point registers is configurable, using the `-fp_num_regs` option. The ARCv3 ABI defines the usage for floating point registers (FPRs).

Register	ABI Function
f0	FP function result; FP function argument 1
f1 to f7	FP function arguments 2 – 8
f8 to f19	Caller-saved; may be freely used by compiler
f20 to f31	Callee-saved; general-purpose registers

The callee-saved FPRs can be saved and restored using the ENTER and LEAVE instructions (note these are 32-bit encoded). These impose a hard constraint that the callee-saved region of the FP register file is always a contiguous set of FPRs at the high-end of the register map. To minimize the number of FPRs saved and restored by function prolog/epilog sequences, compilers must allocate callee-saved FPRs from f31 downwards (or from f15 when 16 FPRs are configured, or from f7 when 8 FPRs are configured).

6.1.3 Multiple Register Banks

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

When 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 (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 DEBUGI.RB field; any write of an index to a non-existent bank will be saturated to the highest available bank.

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

6.1.3.1 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.

6.1.3.2 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.

6.1.4 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 124.

6.1.5 Pointer Registers, GP (r30), FP (r27), SP (r28)

The ARCv3 application binary interface (ABI) defines the following pointer registers: Global Pointer (GP), Frame Pointer (FP), and Stack Pointer (SP), which use registers r30, 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 6-2.

6.1.6 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).

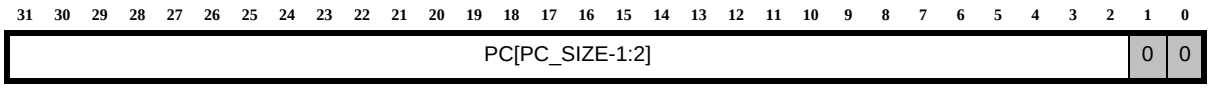
If the delay-slot instruction for any of the branch or jump instructions that set BLINK (BL.D, BLcc.D, JL.D, or JLcc.D) uses a long-immediate operand, the link value assigned to BLINK by the preceding branch or jump-and-link instruction becomes unpredictable. It is illegal for an instruction appearing in the delay-slot of a branch or jump instruction to have a long-immediate data operand (LIMM). 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.

6.1.7 Immediate Data Indicator, LIMM (r62, r60)

The r60 register indicates a 32-bit LIMM that is to be sign-extended to a 64-bit indicates a 32-bit LIMM that is to be zero-extended to a 64-bit value. You can use either r60 or r62 to indicate a null destination.

6.1.8 Word-aligned Program Counter, PCL (r63)

Figure 6-4 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.

6.2 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.

6.3 Extension Core Registers

The core register set is extensible in register positions 32 through 57 (r32-r57). The extension core register r58 is included in an ARC64 core to implement the 64-bit accumulator register (ACC0), although it can also be used as a regular register operand.

Register r58 is reserved and cannot be defined by APEX. However, this register is accessible (explicitly) by APEX.

Register r58 is used as an implicit source and destination accumulator operand of some multiply-accumulate instructions (for example, MACD). However, if r58 is specified as an explicit destination register for an instruction that also writes to it as an implicit accumulator, the implicit write takes precedence over the explicit write and in this case the explicit result value is discarded.

6.3.1 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 122.

6.4 Auxiliary Register Set

A small selection of auxiliary registers are baseline registers, and are therefore present in all ARCv3 systems. Other auxiliary registers are configurable and are present only if their defining extension option is included in the architectural configuration selected at build-time. For a detailed list of the ARCv3 register set for your processor, see "Build and Auxiliary Register List" on page 69.

6.4.1 Baseline Auxiliary Registers

Table 6-2 lists the baseline set of auxiliary registers. The access code definitions of these registers are explained in Table 6-3.

Table 6-2 Baseline Auxiliary Register Set

Number	Auxiliary Register Name	Description
Current Architectural State		
0x4	Core Identity Register, IDENTITY	Processor identification register
0x6	Program Counter, PC	PC register (32-bit)
0x0A	Status Register, STATUS32	Status register (32-bit)
0x412	Branch Target Address, BTA	Branch target address
0x403	Exception Cause Register, ECR	Exception cause register

Table 6-2 Baseline Auxiliary Register Set (Continued)

Number	Auxiliary Register Name	Description
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

6.4.2 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 6-3. 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 6-3 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 6-3, "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 6-3 Key to Auxiliary Register Access Permissions for LR and SR Instructions

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

6.4.3 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.

Similarly, when the `CODE_DENSITY is configured`, the `JLI_BASE` registers are included because they provide an implicit operand to the JLI_S respectively.

Table 6-4 Indexed Table Auxiliary Register (CODE_DENSITY == 1)

Address	Auxiliary Register Name	Description
0x290	Jump and Link Indexed Base Address, JLI_BASE	Jump and link indexed base address

6.4.4 User Extension Auxiliary Registers

The ARCv3 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. For information about the range of auxiliary addresses available for use as extension auxiliary registers, see the *APEX databook*.

The ARCv3 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 6-5 XPU, User Extension Permission Auxiliary Register (APEX_OPTION == 1)

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

When the ARCv3 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 ARCv3-based processor.

Table 6-6 Extension Flags Register, XFLAGS

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

6.4.5 Optional Build Configuration Registers

The ARCv3 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.

6.4.6 Auxiliary Registers Holding Address Values

A significant number of the proprietary auxiliary registers contain address values. When an ARCv3 based processor is configured with a memory management unit (MMU) those addresses can be virtual or physical. The ARCv3 architecture specifies that all address values in auxiliary registers are to be considered virtual unless they are explicitly defined below as containing physical addresses.

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

- Exception Fault Address: "EFA" on page 149
- MMU API registers
- Instruction cache registers: "Invalidate Instruction-Cache Start Region, IC_IVIR" on page 953 and "Invalidate Instruction-Cache End Region, IC_ENDR" on page 955
- Data cache registers: "Data-Cache Region Start Address, DC_STARTR" on page 985 and "Data-Cache Region End Address, DC_ENDR" on page 986
- All MPU base addresses (an MPU can be configured only in systems without MMU and are hence implicitly physical)
- All ARConnect auxiliary registers

6.4.7 Core Identity Register, IDENTITY

Address: 0x04

Access: R

Figure 6-5 IDENTITY Register

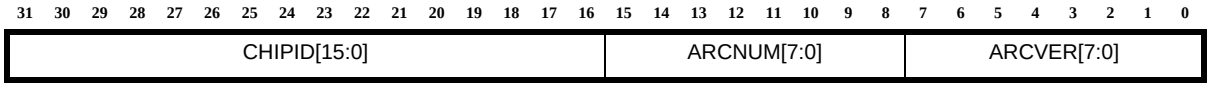


Figure 6-5 shows the identity register (IDENTITY). This register contains the following fields:

Table 6-7 IDENTITY Field Descriptions

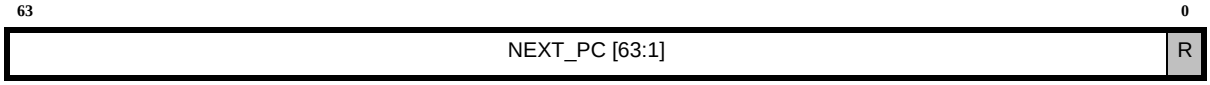
Field	Bit	Description
ARCVER	[7:0]	■ 0x60 to 0x7F = ARC HS core based on the ARCv3 processors - 0x60 - 0x6F: ARC HS5x cores supporting the ARC32 ISA - 0x70 - 0x7F: ARC HS6x cores supporting the ARC64 ISA ■ 0x80 to 0xFF = Reserved
ARCNUM	[15:8]	This field allows you to uniquely identify each core in a multi-core system. Values: `0` to `255` Default: 0 The value of this field is a reflection of the `-arc_num` option in the ARChitect tool. In a multi-core configuration, ARChitect assigns the value of the `-arc_num` option to the first core. All the other cores are assigned increasing sequential values based on their order in the chain. By default, the `-arc_num` option of the first core in the chain is assigned 0 and the other cores in the configuration are assigned core IDs 1, 2, 3, and so on. The input pin `arcnum[7:0]` is provided on the core interface to define the ARCNUM field. Additionally, boot software can use the 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.
CHIPID	[31:16]	This field is deprecated. This field reads 0.

6.4.8 Program Counter, PC

Address: 0x06

Access: r

Figure 6-6 PC Register Addressing Full 64-bit Address Space



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.

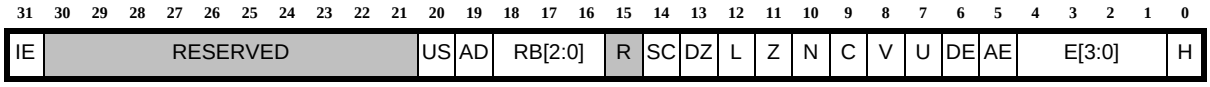
When the configuration option `-pipe_edc_option==true`, any single or double-bit errors in this register are detected, and the signal `edc_pipe_err` is asserted high. At reset, this register is initialized along with EDC code.

6.4.9 Status Register, STATUS32

Address: 0x0A

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

Figure 6-7 STATUS32 Register



The status register, STATUS32 that performs the following functions:

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

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

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

When the ARCv3-based processor reads the status register in user mode, only the Z, N, C, and V bits are visible; all other bits read as zero (RAZ). In kernel mode, all bits of the STATUS32 register are visible. When the processor writes the status register in user mode, only the Z, N, C, and V bits are modified; all other bits are ignored on write (IOW). In kernel mode, in addition to the Z, N, C, and V bits, the US, H, SC, AD, and DZ bits of the STATUS32 register can be modified using the FLAG instruction. A privileged version of the FLAG instruction, called KFLAG, also allows a kernel-mode process to modify the IE, AE, and RB bits.

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

Table 6-8 STATUS32 Bit-Field Definitions and Read / Write Accessibility

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
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 ARCv3 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.

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 103 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.

US

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

When `US==0`:

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

When `US==1`:

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

E[3:0]

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

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

Note For more information about interrupt priority levels, see Exception and Interrupt Priority.

AE

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

DE

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

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

ES

This bit is reserved.

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.

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.

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.

DZ

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

H

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

STATUS32 Behavior on Interrupt Entry or Exit

On interrupt entry, the STATUS32 register is either pushed to the stack or copied to STATUS32_P0, depending on `FIRQ_OPTION` and the level of the interrupt (P0 or otherwise). On exception entry, the STATUS32 register is copied to ERSTATUS. However, an EV_Trap exception clears the ES and DE fields

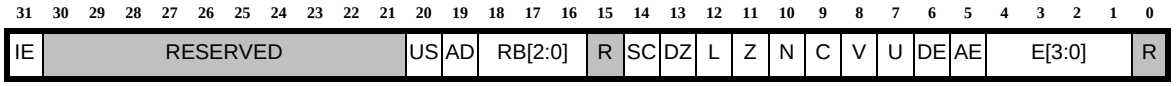
before copying the STATUS32 register to Exception Return Status, ERSTATUS, as EV_Trap is a post-commit exception. The DE, ES, and DZ bits are all cleared when an interrupt or exception is taken, after the STATUS32 register has been saved.

6.4.10 Status Register Priority 0, STATUS32_P0

Address: 0x0B

Access: RW

Figure 6-8 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.

6.4.11 Saved User Stack Pointer, AUX_USER_SP

Address: 0x0D

Access: RW

Figure 6-9 AUX_USER_SP Register

You can read and write to the AUX_USER_SP register in kernel mode; this register is protected and inaccessible in user mode. This register is used to save the kernel stack pointer while in user mode.

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

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

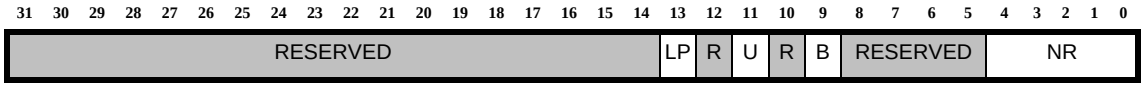
6.4.12 Interrupt Context Saving Control Register, AUX_IRQ_CTRL

Address: 0x0E

Access: RW

Reset value: 0x00000000

Figure 6-10 Table 6-1



The AUX_IRQ_CTRL register controls the behavior of the interrupt prolog and epilog sequences which perform automated register save and restore on interrupt entry and interrupt exit.

Table 6-9 AUX_IRQ_CTRL Field Description

Field	Bit	Description
NR	[4:0]	Indicates number of general-purpose register pairs saved, from 0 to 8 or 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 6-1.
B	[9]	Indicates whether to save and restore BLINK (ignored if NR is greater than or equal to 16.)
U	[11]	Indicates if user context is saved to user stack
LP	[13]	Indicates whether to save and restore JLI_BASE auxiliary registers

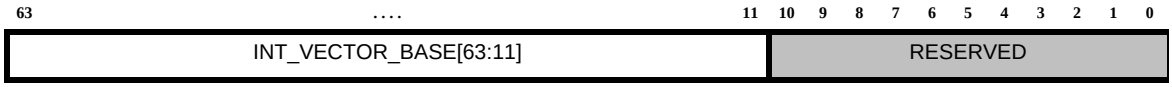
This register is read as zero and ignored on write if fast interrupts are present and only one priority level (P0) is configured. A fast interrupt with only one priority level does not emit a prolog or epilog sequence, so you need not program the set of saved and restored registers. This register is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

6.4.13 Interrupt Vector Base Register, INT_VECTOR_BASE

Address: 0x25

Access: RW

Figure 6-11 INT_VECTOR_BASE Register for ARC64



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 VECBASE_AC_BUILD register by the reset sequence. During program execution, the interrupt vector base can be changed by writing to INT_VECTOR_BASE register.

An ARC64 vector table must be aligned to a 2 kB boundary.

Any bits numbered greater than PC_SIZE - 1 are not implemented and are considered to be reserved in the INT_VECTOR_BASE registers.

Reading any reserved bits returns 0 and writes to such bits have no effect.

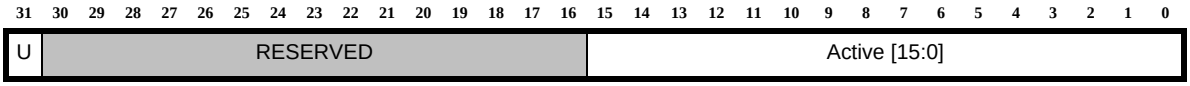
When an MMU is configured, the INT_VECTOR_BASE register must be set to an address in the non-translated memory space. This design allows the vector table to be accessed when taking an MMU fault exception (that is, EV_IMMUFault or EV_DMMUFault).

6.4.14 Active Interrupts Register, AUX_IRQ_ACT

Address: 0x43

Access: RW

Figure 6-12 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 185.

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 6-10 AUX_IRQ_ACT Field Description

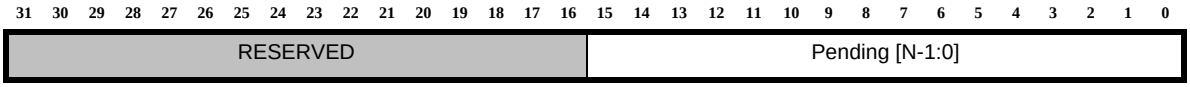
Field	Bit	Description
Active	[15:0]	Bit i indicates whether there is an active interrupt at priority i. If fewer than 16 priority levels are configured, unused bits are read as zero and ignored on write.
U	[31]	Snapshot of the STATUS32.U bit when an interrupt is taken at a point where Active[15:0] == 0.

6.4.15 Interrupt Priority Pending Register, IRQ_PRIORITY_PENDING

Address: 0x200

Access: R

Figure 6-13 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 6-11 IRQ_PRIORITY_PENDING Field Description

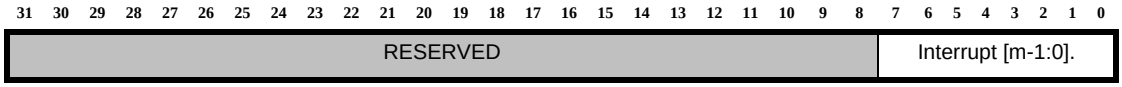
Field	Bit	Description
P	[N-1:0] Where N is the number of interrupt priority levels configured.	Bit i indicates whether there is a pending normal interrupt at priority level i. If fewer than 16 priority levels are configured, the unused bits are read as zero.

6.4.16 Software Interrupt Trigger, AUX_IRQ_HINT

Address: 0x201

Access: RW

Figure 6-14 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 ARCv3-based code or from the host (see "The Host" on page 863).

The software triggered interrupt mechanism can be used even if there are no associated interrupts connected to the ARCv3-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.

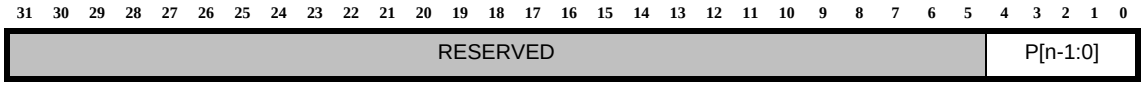
The width of the interrupt fields is determined by the number of interrupts configured in the system. If `M` is the number of interrupts configured, $m=ceil(\log_2(M + 16))$, where ceil is the function to round to the next higher integer.

6.4.17 Interrupt Priority Register, IRQ_PRIORITY

Address: 0x206

Access: RW

Figure 6-15 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. If an interrupt priority > N-1 is assigned, the associated interrupt priority is `N-1`.

The width of the priority field is determined by the number of interrupt priority levels configured in the system. If N is the number of interrupt priority levels configured, $n=ceil(\log_2(N))$, 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 6-12 IRQ_PRIORITY Field Description

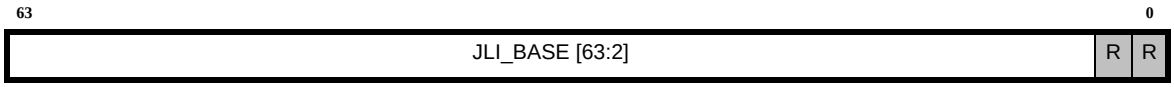
Field	Bit	Description
P	[n-1:0] If N is the number of interrupt priority levels configured, $n=ceil(\log_2(N))$	Value of IRQ_PRIORITY, for the selected interrupt.

6.4.18 Jump and Link Indexed Base Address, JLI_BASE

Address: 0x290

Access: rw

Figure 6-16 JLI_BASE Register when PC_SIZE == 64



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

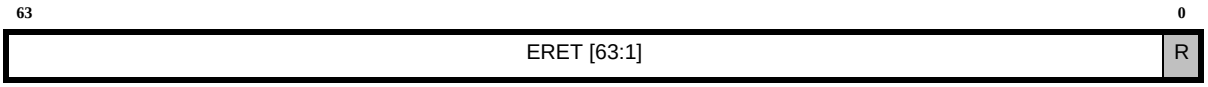
The width of the JLI_BASE register is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

6.4.19 Exception Return Address, ERET

Address: 0x400

Access: RW

Figure 6-17 ERET Register



When returning from an exception, the program counter (see Program Counter, PC) is loaded from the Exception Return Address (ERET) register. ERET is assigned a new value whenever an exception or interrupt is taken. 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 7-4.

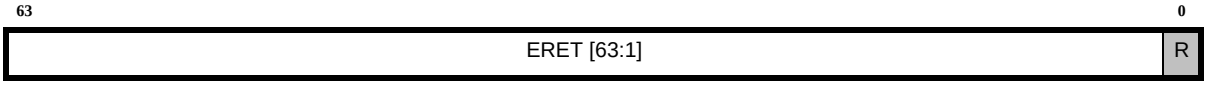
In the ARCv3-based processor, Bit 0 is ignored and must always be set to zero. When read, Bit 0 returns zero.

6.4.20 Exception Return Branch Target Address, ERBTA

Address: 0x401

Access: RW

Figure 6-18 ERBTA Register



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.

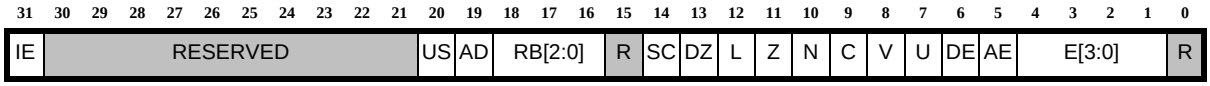
The width of the ERBTA register is determined by the PC size. Reading any reserved bits returns 0 and writes to such bits have no effect.

6.4.21 Exception Return Status, ERSTATUS

Address: 0x402

Access: RW

Figure 6-19 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.

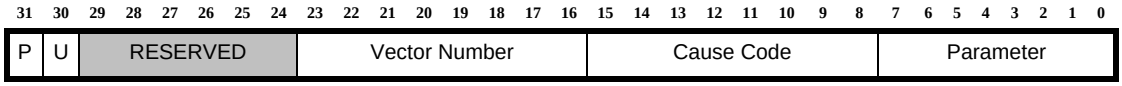
Bit 0 is ignored and must always be set to zero. When read, Bit 0 always returns zero.

6.4.22 Exception Cause Register, ECR

Address: 0x403

Access: RW

Figure 6-20 ECR Register



The Exception Cause register (ECR) provides information about the source of the most recent exception to have been taken. Figure 6-20 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 7-2 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 ARCv3 processor in Appendix A, "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 ARCv3 processor in Appendix A, "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. For more information on how the U bit is set for the relevant type of ARCv3 processor, see either Appendix A.1.1, "ECR User mode Setting in the ARC EM Family of Cores" or Appendix A.3.1, "ECR User mode Setting for the ARC HS6x/EV/VPX5 Family of Cores" .

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.

6.4.23 Exception Fault Address, EFA

Address: 0x404

Access: RW

Figure 6-21 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 7-4.

The address captured in the EFA register, during an exception or by an auxiliary write, is limited in width to ADDR_SIZE.

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:

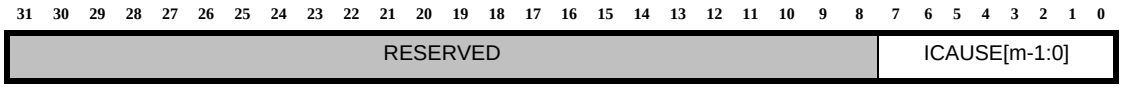
- A bus error exception caused by an uncached load or store operation sets EFA to the physical address of the faulting memory location. EFA is always virtual address for the instruction fetch related memory exceptions.
- When an internal instruction memory error exception occurs, the EFA contains the 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 will include the faulty bit(s), and therefore the tag field within the EFA value cannot be trusted as the copy-back operation will not have taken place. If an ECC or parity error is detected when reading data from the data cache while copying a line back to memory, the value assigned to EFA will be correct.
  - If the exception was due to a bus error reported during a copy-back bus write transaction, then EFA indicates the physical address of the cache line that was being copied back.

6.4.24 Interrupt Cause Registers, ICAUSE

Address: 0x40A

Access: R

Figure 6-22 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=ceil(log2(M + 16)),` where ceil is the function to round to the next higher integer.

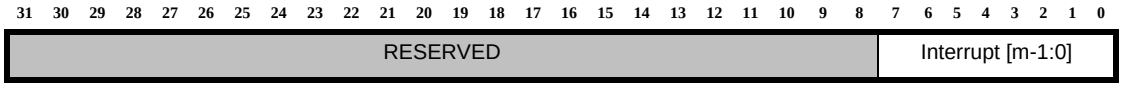
On reset, this register contains `0x00000001` for configured interrupts; if Interrupt [m-1:0] is greater than `M+16-1 or less than 16`, this register is read as zero. Where, `M` is the number of interrupts configured, `m=ceil(log2(M + 16))`, and ceil is the function to round to the next higher integer.

6.4.25 Interrupt Select, IRQ_SELECT

Address: 0x40B

Access: RW

Figure 6-23 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, $m=ceil(\log_2(M + 16))$, 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 6-13 IRQ_SELECT Field Description

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

6.4.26 Interrupt Enable Register, IRQ_ENABLE

Address: 0x40C

Access: RW

Figure 6-24 IRQ_ENABLE Register

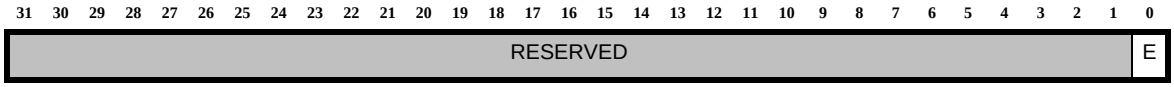


Table 6-14 IRQ_ENABLE Field Description

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

This banked register allows software to enable or disable an interrupt selected by the Interrupt Select, IRQ_SELECT register.

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=ceil(log`$_2$`(M + 16)),` and ceil is the function to round to the next higher integer.

6.4.27 Interrupt Trigger Register, IRQ_TRIGGER

Address: 0x40D

Access: RW

-

Figure 6-25 IRQ_TRIGGER Register

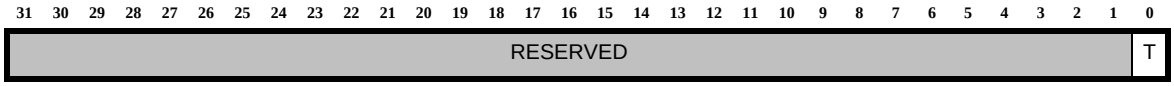


Table 6-15 IRQ_TRIGGER Field Description

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

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

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 to the core.

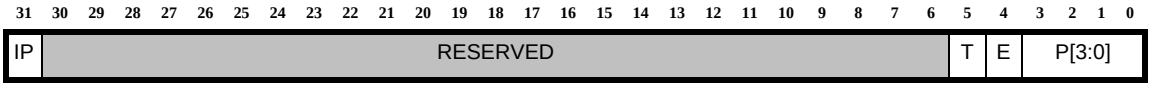
On reset, this register contains `0x00000000` for configured interrupts; if Interrupt [m-1:0] is greater than `M+16-1 or less than 16`, this register is read as zero and ignored on write. Where, `M` is the number of interrupts configured, $m=ceil(\log_2(M + 16))$, and ceil is the function to round to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

6.4.28 Interrupt Status Register, IRQ_STATUS

Address: 0x40F

Access: R

Figure 6-26 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 6-16 IRQ_STATUS Field Description

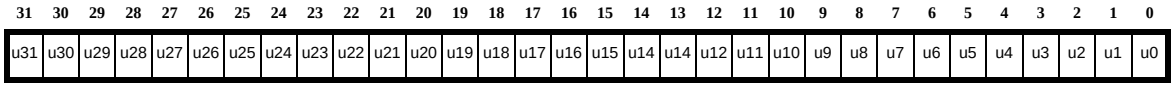
Field	Bit	Description
P	[3:0]	Value of Interrupt Priority Register, IRQ_PRIORITY, for the selected interrupt
E	[4]	Value of Interrupt Enable Register, IRQ_ENABLE, for the selected interrupt
T	[5]	Value of Interrupt Trigger Register, IRQ_TRIGGER, for the selected interrupt
IP	[31]	Value of Interrupt Pending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

6.4.29 User Mode Extension Enable Register, XPU

Address: 0x410

Access: RW

Figure 6-27 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

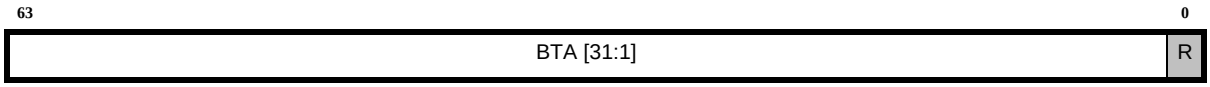
On Reset, the XPU register is set to 0x00000000, so that all extension groups are initially protected from User mode access. For more information about the allocation of XPU bits to each extension group, see the processor-specific Databook.

6.4.30 Branch Target Address, BTA

Address: 0x412

Access: RW

Figure 6-28 BTA Register



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 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.

6.4.31 Interrupt Pulse Cancel Register, IRQ_PULSE_CANCEL

Address: 0x415

Access: W

Figure 6-29 IRQ_PULSE_CANCEL Register

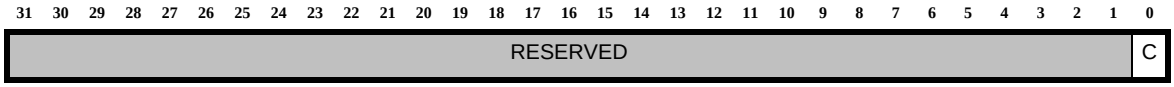


Table 6-17 IRQ_PULSE_CANCEL Field Description

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

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

Reset value is not applicable, as this register cannot be read. A write of pulse-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 to the core.

6.4.32 Interrupt Pending Register, IRQ_PENDING

Address: 0x416

Access: R

Figure 6-30 IRQ_PENDING Register

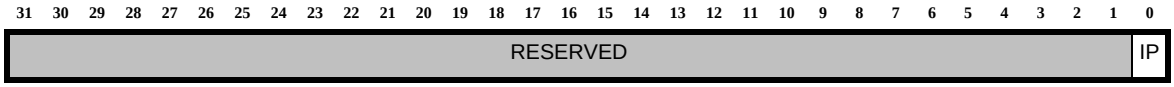


Table 6-18 IRQ_PENDING Field Description

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

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

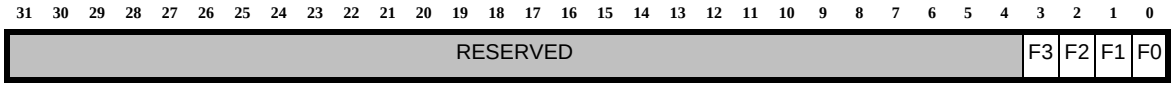
On reset, this register contains `0x00000000` for configured interrupts; if Interrupt [m-1:0] is greater than `M+16-1 or less than 16`, this register is read as zero. Where, `M` is the number of interrupts configured, `m=ceil(log2(M + 16)),` and ceil is the function to round to the next higher integer.

6.4.33 User Extension Flags Register, XFLAGS

Address: 0x44F

Access: RW

Figure 6-31 User Extension Flags Register, XFLAGS



The User Extension Flags register (XFLAGS) is present in a build of an ARCv3-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.

6.5 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 ARCv3-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 ARCv3 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: theleast 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 or absence of the component within the ARCv3-based system.

Auxiliary registers, in the range 0x60 to 0x7F, 0xC0 to 0xFF, and 0xF60 and 0xFFF are assumed to be BCRs. In kernel mode, any read from a non-existent build configuration register in these ranges 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 6-19 summarizes the build configuration registers for components that are described in this document.

Table 6-19 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	ISA Profile Register, ISA_PROFILE	R
0xF9	Micro Architecture Configuration Register, MICRO_ARCH_BUILD	R

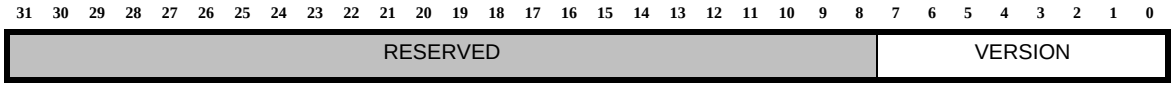
6.5.1 Build Configuration Registers Version, BCR_VER

Address: 0x60

Access: R

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

Figure 6-32 BCR_VER Register



BCR_VER register indicates the regions of the build-configuration registers.

Table 6-20 BCR_VER Field Descriptions

Field	Bit	Description
VERSION	[7:0]	**Version of Build Configuration Registers** ■ 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 0x60-0x7F, 0xC0-0xFF,and 0xF60 – 0xFFF The other values for this field are reserved.

6.5.2 BTA Configuration Register, BTA_LINK_BUILD

Address: 0x63

Access: R

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

Figure 6-33 BTA_LINK_BUILD Register

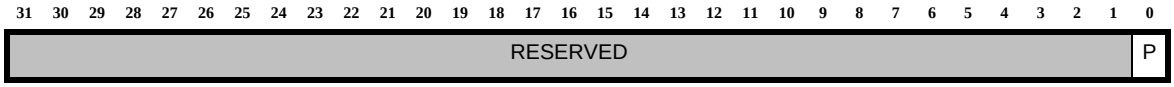


Table 6-21 BTA_LINK_BUILD Field Descriptions

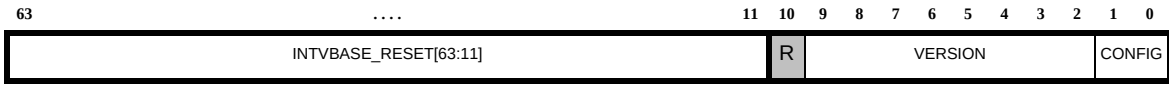
Field	Bit	Description
P	[0]	**Presence of BTA Registers** 0x0 = BTA_L1 and BTA_L2 registers are absent All other values are reserved.

6.5.3 Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD

Address: 0x68

Access: R

Figure 6-34 VECBASE_AC_BUILD Register for ARC64



The default base address of the interrupt vector table is fixed when a particular ARCv3-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. The VECBASE_AC_BUILD register therefore indicates to software the value of the default interrupt vector base address, and how also it was defined.

When the option `-intvbase_ext` is `false`, the reset value of INT_VECTOR_BASE is defined by a build-time constant value provided by the `-intvbase_preset` option. Alternatively, when the `-intvbase_ext` option is `true`, the INT_VECTOR_BASE register is loaded on reset with the `intvbase_in` input port's current value, allowing external hardware to define the initial reset vector when the core is reset. After reset, the software may re-program the INT_VECTOR_BASE register.

The interrupt vector table is always aligned to an address that is a multiple of the maximum size of the table. ARC64 vector tables contain up to 256 entries, each 8 bytes in size, and hence ARC64 vector tables are aligned at 2kB boundaries and the lowest 11 bits of their base address are always zero. For this reason, bit 10 of VECBASE_AC_BUILD is reserved in ARC64 cores and always reads as zero.

Table 6-22 VECBASE_AC_BUILD Field Descriptions

Field	ARC64 Configuration Bit	Description
CONFIG	[1:0]	Configuration of the reset value for INT_VECTOR_BASE: `0x0`: Address is supplied with the `-intvbase_preset` option, when `-intvbase_ext` is false. `0x1`: Address is supplied through the `intvbase_in` core input signal, when the `-intvbase_ext` option is true. Other values are reserved.
VERSION	[9:2]	Version of Interrupt Unit 0x04 = ARCv2Interrupt Unit 0x05 = ARCv3 Interrupt Unit
INTVBASE_RESET	[63:11]	The upper bits of the INTVBASE_RESET value, excluding the lower 11 bits for ARC64 cores, which are assumed to be zero.

6.5.4 Core Register File Configuration Register, RF_BUILD

Address: 0x6E

Access: R

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

Figure 6-35 RF_BUILD Register

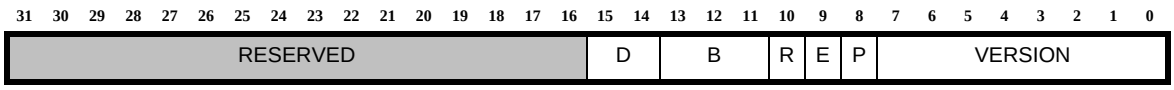


Table 6-23 RF_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	**Version of Core Register Set** 0x02 = 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
R	[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 6-23 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.

6.5.5 Multiplier Configuration Register, MULTIPLY_BUILD

Address: 0x7B

Access: R

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

Figure 6-36 ARC64 MULTIPLY_BUILD Register

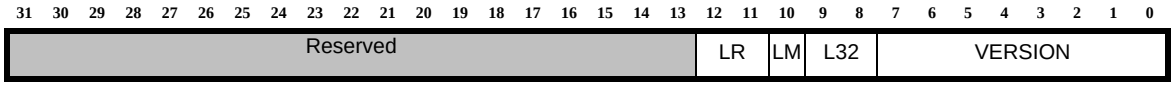


Table 6-24 ARC64 MULTIPLY_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	**Version of Multiply Support** 0x10 = ARC64 default value
L32	[9:8]	Latency of 32x32 multiply instructions ■ 0x2 = 3 cycles
LM	[10]	Indicates support for 64x64 multiply instructions. Note: If configuration option `-mpy64` is set to true, then LM = 1, otherwise LM = 0.
LR	[12:11]	Repeat interval for 64x64 multiply instructions ■ 0x3 = blocking, with four-cycle repeat interval

6.5.6 Swap Instruction Configuration Register, SWAP_BUILD

Address: 0x7C

Access: R

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

Figure 6-37 SWAP_BUILD Register

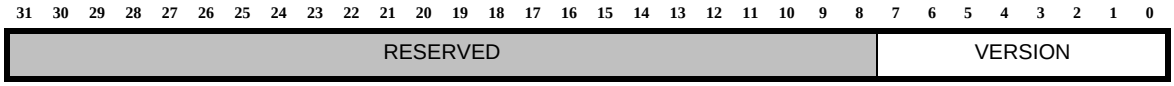


Table 6-25 SWAP_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	**Version of Swap** 0x01 = ARCompact (ARCv1) version of SWAP option 0x03 = ARCv3 version of SWAP option

6.5.7 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 6-38 NORM_BUILD Register

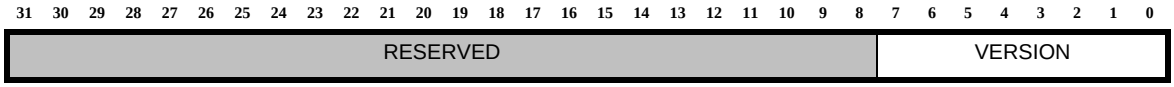


Table 6-26 NORM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	**Version of Norm** 0x01 = Reserved 0x02 = ARCompact (ARCv1) version 0x03 = ARCv3 version

6.5.8 Min/Max Instruction Configuration Register, MINMAX_BUILD

Address: 0x7E

Access: R

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

Figure 6-39 MINMAX_BUILD Register

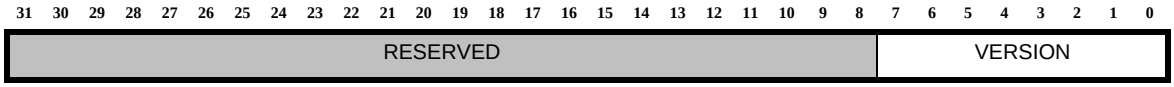


Table 6-27 MINMAX_BUILD field descriptions

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

6.5.9 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 6-40 BARREL_BUILD Register

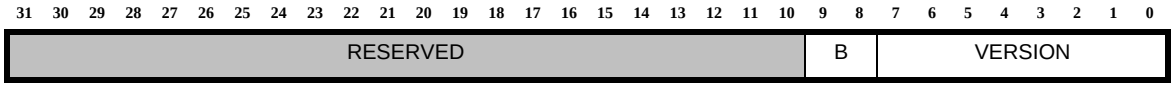


Table 6-28 BARREL_BUILD Field Descriptions

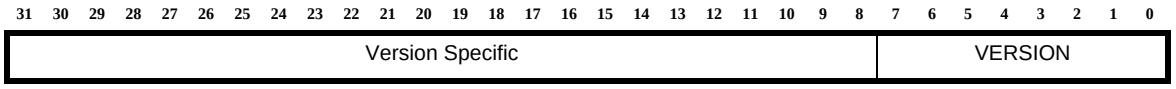
Field	Bit	Description
VERSION	[7:0]	**Version of Barrel Shifter** 0x01 = Reserved 0x02 = ARCompact (ARCv1) version 0x03 = ARCv3 version
B[1:0]	[9:8]	**SHIFT_OPTION** 0 = Single-bit shifts only 1 = Single-bit and shift-assist instructions 2 = Single-bit and barrel-shift instructions 3 = All shift instructions

6.5.10 Instruction Set Configuration Register, ISA_CONFIG

Address: 0xC1

Access: R

Figure 6-41 ISA_CONFIG Build Register



The common fields of the ISA_CONFIG registers are defined in ISA_CONFIG Field Descriptions.

Table 6-29 ISA_CONFIG Field Descriptions

Field	Bit	Description
VERSION	[7:0]	0x01 = ARCompact (ARCv1) version 0x02 = ARCv2 configuration 0x03 = ARC32 configuration of ARCv3 0x04 = ARC64 configuration of ARCv3
Version Specific	[31:8]	ISA configuration fields defined according to the VERSION field

Figure 6-42 ISA_CONFIG BCR for Version 0x04

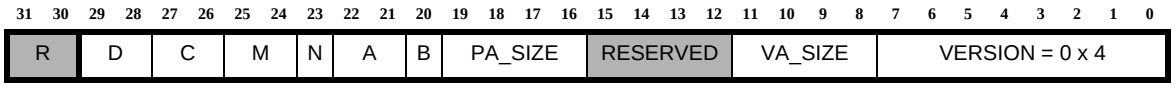


Table 6-30 explains the version 0x4 field descriptions of the ISA_CONIFIG register.

Table 6-30 ISA_CONFIG Fields for ARC64 Version

Field	Bit	Description
VA_SIZE	[11:8]	■ 0x0: 40-bit virtual address (VA), 64-bit VA defined as { {24{VA[39]}}, VA[39:0] } ■ 0x1: 48-bit virtual address, 64-bit VA defined as { {16{VA[47]}}, VA[47:0] } ■ 0x2: 56-bit virtual address, 64-bit VA defined as { {8{VA[55]}}, VA[55:0] } ■ 0x3: 64-bit virtual address All other values reserved
PA_SIZE	[19:16]	■ 0x0 : 40-bit physical address ■ 0x1 : 48-bit physical address ■ 0x2 : 56-bit physical address ■ 0x3 : 64-bit physical address All other values reserved

Table 6-30 ISA_CONFIG Fields for ARC64 Version (Continued)

Field	Bit	Description
B	[20]	0x0 : Little-endian byte ordering 0x1 : Big-endian byte ordering
A	[22:21]	■ 0x0: Baseline atomic memory operations are supported, that is EX and EX.DI ■ 0x1: In addition to the instructions supported by field A==0, the following instructions are supported: LLOCK, LLOCK.DI, SCOND, SCOND.DI, WLFC, WAIT and any 64-bit versions of these instructions when `-ll64_option` is true ■ 0x2: In addition to the instructions supported by field A==1, the following instructions are supported: ex<.aq.rl>, atld.add<.aq.rl>, atld.or<.aq.rl>, atld.and<.aq.rl>, atld.xor<.aq.rl>, atld.minu<.aq.rl>, atld.maxu<.aq.rl>, atld.min<.aq.rl>, atld.max<.aq.rl>
N	[23]	■ 0x0 : All data memory references must use natural alignment ■ 0x1 : Non-aligned memory references are supported
M	[25:24]	0x0 : 128-bit load/store operations not supported (`-m128_option = false`) 0x1 : 128-bit load operations are supported (`-ld128_option = true`, and `-m128_option = false`) 0x2: 128-bit load and store operations are supported (`-m128_option = true`) 0x3: Reserved
C	[27:26]	0x3 : ISA_PROFILE register configuration determines the code density instructions that are included in the processor. All other values are reserved
D (DIV_REM_OPTION)	[29:28]	■ 0x3 : radix-4 fast divider implements 32-bit and 64-bit integer DIV/REM instructions (DIV, DIVL, DIVU, DIVUL, REM, REML, REMU, and REMUL). This is the default baseline value. All other values are reserved

6.5.11 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 6-43 IRQ_BUILD Register

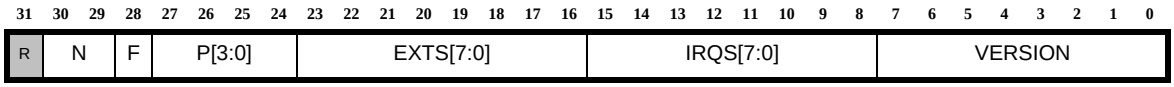


Table 6-31 IRQ_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	**Version of the Interrupt Controller** 0x01 = ARC v2 0x02 = Indicates support for the multi-core NMI feature.
IRQS	[15:8]	Indicates the number of external interrupt lines configured in the Interrupt Controller.
EXTS	[23:16]	Indicates the minimum number of private external interrupt lines configured in the Interrupt Controller, as indicated by `-external_interrupts` option.
P	[27:24]	Contains N-1, when N interrupt priority levels are configured.
F	[28]	Value of the FIRQ_OPTION configuration option.
N	[29]	Indicates whether the non-maskable imprecise exception is supported. ■ 0x0: not supported. ■ 0x1: single-core non-maskable imprecise exception is supported. ■ 0x2: multi-core cluster-level non-maskable imprecise exception is supported. The following registers are added to the processor when this option is configured. This bit reflects the value of the `-nmi_option` configuration option.

Note

The reset values of this register is defined by the configuration of the interrupt controller, which sets the number of interrupts and the number of those interrupts that are externally visible as interrupt lines.

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.

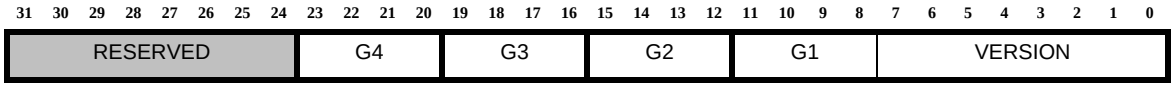
If the non-maskable imprecise exception is configured with a cluster component, but the number of configured interrupts is 0, then the IRQ_BUILD register contains the value 0x40000002.

6.5.12 ISA Profile Register, ISA_PROFILE

Address: 0xF8

Access: R

Figure 6-44 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 6-32 describes the three opcode groups that can be configured in a processor.

Table 6-32 ISA_PROFILE Field Descriptions

Field	Bit	Description
VERSION	[7:0]	■ 0x1: Previous ISA_PROFILE format (ARCv2) ■ 0x2: Current ISA_PROFILE format (ARCv3)
G1	[11:8]	■ 0x0: No extension group is included ■ 0x1: Extension Group CD1a is included: - F16_LD_ADD_SUB - LD_ST_R01 ■ 0x2 to 0xF: Reserved
G2	[15:12]	■ 0x0: No extension group is included ■ 0x1: Extension Group CD2 is included: - F16_LD_BYTE - F16_LD_HALF - F16_LDX_HALF - F16_ST_BYTE - F16_ST_HALF - F16_BL ■ 0x2: Extension Group VDSP is included: - F32_VEC1 - F32_VEC2 - F32_VEC3 - F32_VEC4 - F32_VEC5 - F32_VEC6 ■ 0x3 to 0xF: Reserved

Table 6-32 ISA_PROFILE Field Descriptions (Continued)

Field	Bit	Description
G3	[19:16]	■ 0x0: No extension group is included ■ 0x1: Extension Group CD3 is included: - F16_JLI_EI ■ 0x2: Extension Group ARC64a is included: - F32_GEN_OP64 ■ 0x3 to 0xF: Reserved
G4	[23:20]	■ 0x0: No extension group is included ■ 0x1: Extension Group CD1b is included: - F16_COMPACT1 - F16_SP_OPS - F16_GP_LD_ADD ■ 0x2: Extension Group ARC64b is included: - JLI_S_U10 - F16_SP_OP64 - F16_GP_OP64 - F16_MOVL ■ 0x3 to 0xF: Reserved

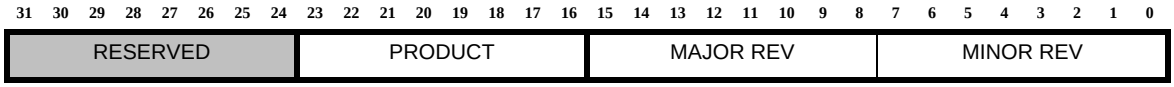
When an ARCv3 core is configured with the ARC64 ISA including all available code density options,the ISA_PROFILE register returns 0x00221102, that is, ARC64 ISA_PROFILE = {0x00, ARC64b, ARC64a, CD2, CD1a, 0x02}.

6.5.13 Micro Architecture Configuration Register, MICRO_ARCH_BUILD

Address: 0xF9

Access: R

Figure 6-45 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. Table 6-33 lists the release revision numbers for each product family:

Table 6-33 Product Family and The Release Revisions

Product Family	Value
ARC HS 6x	0x06_00_00

Table 6-34 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. ■ 0x00: ARC HS6x ■
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. ■ 0x00: ARC HS6x family ■
Product Family	[23:16]	This field indicates the product family. ■ 0x06: ARC HS6x family ■

7 Interrupts and Exceptions

7.1 Introduction

ARCv3-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 floating-point exception and Memory Error exceptions originating from bus errors external to the ARCv3-based system, both of which 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.

7.2 Privileges and Operating Modes

The operating mode of the processor determines whether a task is permitted to execute an instruction, or to access protected state, that requires Kernal privilege. 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 software executing in Kernel mode can examine bits in the STATUS32 register (see Status Register, STATUS32) to determine the current 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.

7.2.1 Kernel Mode

Kernel mode has the highest level of privilege and is the operating mode entered from Reset. Access to complete machine state, including the ability to execute privileged instructions and privileged registers, is provided in kernel mode.

7.2.2 User Mode

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.

7.2.3 Privilege Violations

This section describes the privileges available to the tasks running in user mode and kernel mode. Table 7-1 gives an overview of the differences in privilege between the two modes.

Table 7-1 Overview of Privileges in Kernel and User Modes

Function	User Mode	Kernel Mode
Access to General Purpose Registers	All, except ILINK for which no access is allowed in user mode	Yes
Memory management / TLB controls	No	Yes
Cache management	No	Yes
Access to memory with ASID = User PID	By flag bits in Page Descriptor (PD)	By flag bits in Page Descriptor (PD)
Access to memory with ASID ≠ User PID	If global bit set	If global bit set
Unprivileged instructions	Yes	Yes
Privileged instructions	No	Yes
Access to Basecase Auxiliary Registers	Only PC and STATUS32[ZNCV]	Yes
Access to extension auxiliary registers	JLI_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

7.2.3.1 Privileged Instructions

All ARCv3 instructions are unprivileged unless specifically defined as privileged. Privileged instructions are:

- SLEEP
- WLFC
- WAIT, when STATUS32[US] = 0
- WEVT
- 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.

7.2.3.2 Privileged Registers

Access to the majority of general-purpose registers is not affected by the operating mode. Switching between 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 User and Kernel modes. The auxiliary registers accessible in User mode include the following:

- Program Counter, PC
- Status Register, STATUS32 - ZNCV flags only

7.2.4 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 will switch 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 User 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).

7.3 Interrupts

The ARCv3 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 ARCv3-based processors. When building a processor, you can configure the processor to include an interrupt unit. The ARCv3 interrupt unit is highly programmable. The ARCv3 interrupt unit supports the following interrupt types:

- Timer — triggered by one of the optional extension timers
- Multi-core interrupts —triggered by one of the cores in a multi-core system
- External — available as input pins to the core
- Software-only — triggered by software only

This section explains the following topics:

- Interrupt Unit Features
- Interrupt Unit Configuration
- Interrupt Unit Programming
- Interrupt Handling

7.3.1 Interrupt Unit Features

ARCv3 interrupt unit has the following interrupt specifications:

- Support for up to 240 interrupts
  - User configurable from 0 to 240
  - Level sensitive or pulse sensitive
- Support for up to 16 interrupt priority levels
  - Programmable from 0 (highest priority) to 15 (lowest priority)
- 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

7.3.2 Interrupt Unit Configuration

See the ARCv3-based processor Databook.

7.3.2.1 Interrupt Vectors

Each vector is a 64-bit address pointing to the appropriate handler for the exception or interrupt. The vectors are stored in the native endianess.

Table 7-2 lists the vector offsets. 64-bit double wordis 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 64-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 53 bits.

7.3.2.2 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.

7.3.3 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.

7.3.3.1 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 "Interrupt Select, 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. Interrupt Priority Register, IRQ_PRIORITY: Program the priority of the selected interrupt by writing the priority `(0 to (NUMBER_OF_LEVELS-1))` to the IRQ_PRIORITY register. By default, IRQ_PRIORITY is set to `0`.

b. Interrupt Trigger Register, 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. Interrupt Pulse Cancel Register, 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. Interrupt Enable Register, IRQ_ENABLE: Write `1` to the IRQ_ENABLE register to enable the interrupt. By default, the IRQ_ENABLE register is set to `1`.

e. Interrupt Pending Register, IRQ_PENDING: This register is read-only. This register indicates if the selected interrupt is pending.

f. Interrupt Status Register, 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 ARCv3 interrupt unit. Ensure that you have the necessary privileges to program the interrupt unit.

7.3.3.2 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.

7.3.3.3 Disabling Interrupts

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

7.3.3.4 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).

7.3.3.5 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 ARCv3 ISA provides up to 16 levels of priority, and allows each configured interrupt to have its interrupt priority set dynamically under software control.

7.3.3.6 Interrupt Prioritization and Preemption

In ARCv3-based processors, Priority 0 (P0) is the highest interrupt priority, and Priority 15 (P15) is the lowest. Current operating interrupt priority defines how the processor reacts when servicing an interrupt or an exception and a new interrupt is activated.

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

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

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

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

1. An external level-sensitive interrupt with an asserted interrupt request signal, or
2. An external pulse-sensitive interrupt with an outstanding interrupt request, or
3. An internal software-triggered interrupt with an interrupt number equal to the value in AUX_IRQ_HINT.

The current operating interrupt priority for a processor is determined by the AUX_IRQ_ACT register. The bit position, 0 - 15, of the least-significant 1 in the lower 16 bits of the AUX_IRQ_ACT register is the current operating interrupt priority. If the lower 16 bits of the AUX_IRQ_ACT register are zero, it indicates that the processor is not currently handling an interrupt.

A processor is interrupted if the following conditions are met:

1. The processor is not halted (STATUS32.H == 0), and
2. Interrupts are globally enabled (STATUS32.IE == 1), and
3. The processor is not currently handling an exception (STATUS32.AE == 0), and
4. If there exists an active interrupt with priority that is higher than (that is, numerically less than) the current operating interrupt priority, and the active interrupt has the same priority or higher than the interrupt preemption threshold (that is, numerically less than or equal to) specified by the STATUS32.E bits.

Hint For more information about the behavior and timing of interrupts, see the ARCv3-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.

7.3.3.7 Banked Interrupt Registers

An interrupt controller supporting *M* interrupts ($1 \le M \le 240$), 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 ≥ *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 Interrupt Select, 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 7-1 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 7-1 Setting Interrupt Priority or Enable (STATUS32.IE) of an Interrupt

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

Note The CLRI and SETI instructions are non-serializing, so the preceding sequence takes 4 clock cycles to execute.

7.3.3.8 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 7-1 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.

7.3.3.9 Interrupt Sensitivity Level Programming

Each interrupt can be programmed to be either pulse-sensitive or level-sensitive.

An interrupting device, set to pulse-sensitive interrupts, must assert the interrupt line only once, and then de-assert the interrupt line. For pulse-sensitive interrupts, at most one interrupt is generated for each positive transition of an interrupt line. You can clear the pending interrupt by writing `1` to the corresponding IRQ_PULSE_CANCEL register.

An interrupting device, set to level-sensitive interrupt, must assert and hold the interrupt line until instructed to de-assert the interrupt line by the corresponding interrupt service routine.

Hint For more information about the behavior and timing of interrupts, see the ARCv3-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.

7.3.3.10 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.

7.3.3.11 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 ARCv3-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.

7.3.3.12 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.

7.3.4 Interrupt Handling

The ARCv3-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.

7.3.4.1 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.

7.3.4.2 Fast Interrupts

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

```
FIRQ_OPTION
```

The following is the behavior of fast interrupt option:

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

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

When FIRQ_OPTION is 1 and RGF_NUM_BANKS >= 2, the interrupt entry sequence sets the register bank number STATUS32.RB to 1, the second register bank, when entering a priority P0 interrupt. Similarly, on P0 interrupt exit, the processor returns STATUS32.RB to its previous value by restoring STATUS32.RB from STATUS32_ P0. This configuration causes an automatic switch of the duplicated register banks, allowing all P0 interrupts to use a private subset of registers.

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

7.3.4.2.1 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).

7.3.4.2.2 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.

Example 7-2 Fast Interrupt Entry

```
EnterFastInterrupt(PI) {
   // Assert ((P != 0) && (STATUS32.IE) && (STATUS32.E ? 1)
   // && (STATUS32.AE == 0) && (STATUS32.DE == 0) && (STATUS32.ES == 0))
   ILINK = PC;
   ERET = PC;
   STATUS32_P0 = STATUS32;
   // save user-mode flag
   if (AUX_IRQ_ACT.ACTIVE == 0){
      AUX_IRQ_ACT.U = STATUS32.U
   }
   AUX_IRQ_ACT.ACTIVE[W] = 1;
   // switch to kernel SP if previously in user thread
   if (STATUS32.U){
      AEX R28,[AUX_USER_SP]
   }
   STATUS32.{Z = U; U = 0; L = 1; ES = 0; DZ = 0; DE = 0;}
   if (RGF_NUM_BANKS > 1){
      STATUS32.{RB=1}
   }
   AcknowledgeInterrupt(PI);
   Jump(InterruptVector(PI));
}
```

7.3.4.2.3 Fast Interrupt Exit

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

Example 7-3 Fast Interrupt Exit

```
ReturnFromFastInterrupt (PI, W) {
   // Assert ((P == 0) && (STATUS32.AE == 0))
   //if another fast interrupt is pending and enabled
   if ((W == 0) && (W ≤ STATUS32.E) && (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);
    }
}
```

7.3.4.3 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.

7.3.4.3.1 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 ≥ 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 Current ABI Register Usage. The hardware save and restore mechanism ignores the AUX_IRQ_CTRL.B bit if NR ≥ 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.

When (`code_density_option==true`), JLI_BASE is 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`, User or Kernel mode stack pointers are exchanged; Hence, 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 Kernel stack when a Kernel thread or interrupt handler is interrupted.
- If `AUX_IRQ_CTRL.U == 0` on interrupt entry, the saved registers are saved to the kernel stack regardless of operating mode and current interrupt level.
- When a user thread is interrupted (or an exception occurs), and `AUX_IRQ_CTRL.U == 1`, the user and kernel stack pointers are switched automatically after registers have been saved to the user stack. On return to a user thread, from either an interrupt or an exception, when `AUX_IRQ_CTRL.U == 1`,

the kernel and user stack pointers are switched automatically before core registers are restored to the user stack. However, when `AUX_IRQ_CTRL.U == 0`, any stack pointer switching takes place before state is pushed, and after state is popped. In this case, the kernel stack is used.

Note When the core is configured with fast interrupts (`firq_option==true`) and more than one register bank is present (`RGF_NUM_BANKS > 1`), software cannot set the `AUX_IRQ_CTRL.U` bit to `0`; that is, the core does not support the option to save interrupt context to the kernel stack when a regular interrupt occurs in the user mode.

Saving and restoring of BLINK (R31) can be controlled independently of the general purpose pairs.

7.3.4.3.2 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 ARCv3 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 ARCv3 application binary interface (ABI) defines the pointers to be 32-bit aligned and ARCv3 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.

7.3.4.3.3 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 7-4 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;
  saved_regs    = ((RGF_NUM_REGS == 16) && (AUX_IRQ_CTRL.NR > 8)) ? 16 : 2*NR;
  save_blink    = ((AUX_IRQ_CTRL.NR < 16) && (AUX_IRQ_CTRL.B == 1)) ? 1 : 0;
  save_jli_base = AUX_IRQ_CTRL.LP ? 1 : 0;

  tmp_addr = SP – (8 * (saved_regs + save_blink + save_jli_base + 2));

  if (RGF_NUM_REGS == 32){
    for (i = 0; i < (2 * AUX_IRQ_CTRL.NR)); i += 2){
      Store(Ri,   tmp_addr); tmp_addr += 8;
      Store(Ri+1, tmp_addr); tmp_addr += 8;
    }
    if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16)){
      Store(BLINK, tmp_addr); tmp_addr += 8;
    }
  } else {                          // (RGF_NUM_REGS == 16)
    if (AUX_IRQ_CTRL.NR >= 1){
      Store(R1, tmp_addr); tmp_addr += 8;
      Store(R0, tmp_addr); tmp_addr += 8;
    }
    if (AUX_IRQ_CTRL.NR >= 2){
      Store(R2, tmp_addr); tmp_addr += 8;
      Store(R3, tmp_addr); tmp_addr += 8;
    }
    if (AUX_IRQ_CTRL.NR >= 3){
      Store(R11, tmp_addr); tmp_addr += 8;
      Store(R10, tmp_addr); tmp_addr += 8;
    }
    if (AUX_IRQ_CTRL.NR >= 4){
      Store(R13, tmp_addr); tmp_addr += 8;
      Store(R12, tmp_addr); tmp_addr += 8;
    }
    if (AUX_IRQ_CTRL.NR >= 5){
      Store(R15, tmp_addr); tmp_addr += 8;
      Store(R14, tmp_addr); tmp_addr += 8;
    }
  }
```

```
  if (AUX_IRQ_CTRL.LP) {
    store(JLI_BASE, tmp_addr); tmp_addr += 8;
  }

  Store(STATUS32, tmp_addr); tmp_addr += 8;
  store(PC, tmp_addr); tmp_addr += 8;
  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.

7.3.4.4 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 ≤ 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.

7.3.4.4.1 Regular Interrupt Exit

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

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 7-5 Regular Interrupt Exit

```
ReturnFromInterrupt(PI, W){
  // Assert(((W >= P) || (W < STATUS32.E)) && ((P > 0) || (FIRQ_OPTION ==0)) &&
  //        (STATUS32.AE == 0))

  // clear the currently-active interrupt
  AUX_IRQ_ACT.ACTIVE[P] = 0;
```

```
// if pending interrupts, avoid pop/push
if (((W > 0) || (FIRQ_OPTION == 0)) && ((W < Q) && (W ? STATUS32.E))) {
  // set new interrupt level
  AUX_IRQ_ACT.ACTIVE[W] = 1;
  STATUS32.Z = U;
  // acknowledge interrupt
  AcknowledgeInterrupt(PI);
  // jump to interrupt vector
  Jump(InterruptVector(PI));
}
// else pop context before interrupt exit
else {
  tmp_addr = SP;
  // 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 a 32-entry register file
  if (RGF_NUM_REGS == 32) {
    // pop AUX_IRQ_CTRL.NR register pairs
    for (i = 0; i < (2 * AUX_IRQ_CTRL.NR); i += 2){
      Load(Ri,   tmp_addr); tmp_addr += 8;
      Load(Ri+1, tmp_addr); tmp_addr += 8;
    }
    // if BLINK should be restored separately
    if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 16)){
      Load(BLINK, tmp_addr); tmp_addr += 8;
    }
  } else { // if the register bank has a 16-entry register file
    if (AUX_IRQ_CTRL.NR >= 1) { Load(R0,  tmp_addr);     Load(R1,  tmp_addr+8);   }
    if (AUX_IRQ_CTRL.NR >= 2) { Load(R2,  tmp_addr+16);  Load(R3,  tmp_addr+24);  }
    if (AUX_IRQ_CTRL.NR >= 3) { Load(R10, tmp_addr+32);  Load(R11, tmp_addr+40);  }
    if (AUX_IRQ_CTRL.NR >= 4) { Load(R12, tmp_addr+48);  Load(R13, tmp_addr+56);  }
    if (AUX_IRQ_CTRL.NR >= 5) { Load(R14, tmp_addr+64);  Load(R15, tmp_addr+72);  }
    if (AUX_IRQ_CTRL.NR >= 6) { Load(R26, tmp_addr+80);  Load(R27, tmp_addr+88);  }
    if (AUX_IRQ_CTRL.NR >= 7) { Load(R28, tmp_addr+96);  Load(R29, tmp_addr+104); }
    if (AUX_IRQ_CTRL.NR >= 8) { Load(R30, tmp_addr+112); Load(R31, tmp_addr+120); }
    tmp_addr += (8 * AUX_IRQ_CTRL.NR);
    if (AUX_IRQ_CTRL.B && (AUX_IRQ_CTRL.NR != 8)) {
      Load(R31, tmp_addr); tmp_addr += 8;
    }
  }
}

// restore Code Density state
if (AUX_IRQ_CTRL.LP == 1) {
  Load(JLI_BASE, tmp_addr); tmp_addr += 8;
}

// pop PC into ILINK
Load(PC, tmp_addr); tmp_addr += 8;
```

```
// pop STATUS32
Load(STATUS32, tmp_addr); tmp_addr += 8;

AUX_IRQ_ACT.U = 0;
SP = tmp_addr;

// 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(ILINK);
```

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.

7.3.4.5 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 ARCv3 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

7.3.4.6 Determine Active Interrupts

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

Example 7-6 Determine Active Interrupts

```
scan_interrupt_vectors()
{
   CLRI R0 ;; disable interrupts before manipulating IRQ_SELECT
   for all interrupt numbers x from 0 to 255
   do {
       write x to the IRQ_SELECT register
       write x to the AUX_IRQ_HINT register
```

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

7.4 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 Machine Check (double-fault) exception.

All interrupts and exceptions cause the processor to 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.

7.4.1 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 precisely knows 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.

ARCv3-based processors implement precise exceptions for all types of exception other than external memory faults and floating-point exceptions (when programmed to allow relaxed exception ordering). All instructions can be restarted up and until point where they are committed. Apart from instructions that retire post-commit, instructions can always be dismissed before completion and restarted later. On receipt of an exception, an operating system can do any of the following:

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

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

- An instruction cache load that causes a bus error, and hence a Machine Check, Internal Instruction Memory Error, is precise because the address of the instruction is known at the time of the memory error.
- A load or store operation that causes a bus error, and hence a Bus Error on Data Access to External Data Memory exception. 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.

The precision with which floating-point (FP) exceptions are raised is programmable via the FP_CTRL register; they can be strictly ordered or imprecise. There is normally a performance penalty for programming strictly ordered exceptions, due to the serialization of FP instructions that is required to ensure strict ordering of exceptions. However, floating-point exceptions that are raised when strict ordering of FP exceptions are disabled will be raised imprecisely.

Imprecise exceptions are events that The precision with which floating-point (FP) exceptions are raised is programmable via the FP_CTRL register; they can be strictly ordered or imprecise. There is normally a performance penalty for programming strictly ordered exceptions, due to the serialization of FP instructions that is required to ensure strict ordering of exceptions. However, floating-point exceptions that are raised when strict ordering of FP exceptions are disabled will be raised imprecisely.

Imprecise exception is raised when the processor is in exception mode, the first such delayed exception is taken after the processor has exited exception mode. Any second or subsequent imprecise exceptions of the same types that are raised while the first imprecise exception is pending are discarded.

7.4.2 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

7.4.2.1 Vector Name

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

7.4.2.2 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.

7.4.2.3 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.

In ARC64 vector table each entry is 8 bytes.

The vector offsets are summarized in Table 7-2.

Table 7-2 Interrupt/Exception Vectors

Name	Vector Number	ARC64 Offset	Exception Types
Reset	0x00	0x00	Exception

Table 7-2 Interrupt/Exception Vectors (Continued)

Name	Vector Number	ARC64 Offset	Exception Types
Memory Error	0x01	0x08	Exception
Instruction Error	0x02	0x10	Exception
EV_MachineCheck	0x03	0x18	Exception
EV_IMMUFault	0x04	0x20	Exception
EV_DMMUFault	0x05	0x28	Exception
EV_ProtV	0x06	0x30	Exception
EV_PrivilegeV	0x07	0x38	Exception
EV_SWI	0x08	0x40	Exception
EV_Trap	0x09	0x48	Exception
EV_Extension	0x0A	0x50	Exception
EV_DivZero	0x0B	0x58	Exception
Unused	0x0C	0x60	Exception
EV_Misaligned	0x0D	0x68	Exception
EV_VecUnit	0x0E	0x70	Vector unit exceptions
Unused	0X0F	0x78	Exception
IRQ16 - IRQ 255	0x10 - FF	0x80 - 0x7F8	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.

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 7-7 Pass Control to an Handler or Perform a Soft Reset

1. Read the contents of a 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 7-8 Exception Vector Code

```
_reset:                   ; entry point
 .long _start             ; exception 0 offset 0x0   0
 .long memory_error       ; exception 1 offset 0x4   4
 .long instruction_error  ; exception 2 offset 0x8   8
```

Table 7-3 provides more information about the exception priorities and exception cause parameters.

7.4.2.4 Cause Codes

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

7.4.2.5 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.

7.4.2.6 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 6-20.

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*.

7.4.3 Exception Types and Priorities

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

Interrupts and exceptions are evaluated with the following priority:

1. Reset
2. Machine Check, Double Fault
3. Machine Check, Fatal TLB errors
    a. Invalid TLB command
    b. Illegal Overlapping MPU entries
4. Interrupt
5. TLB Fault on Instruction Fetch
    a. Machine Check, Memory Error on ITLB Access
    b. ITLB miss
6. Protection Violation on Instruction Fetch (MMU or MPU)
7. Memory Error on instruction fetch:
    a. Bus error accessing external instruction memory
    b. Instruction fetch spanning multiple MPU regions
    c. Instruction fetch spanning multiple instruction memory targets
8. Privilege Violation, Actionpoint hit on Instruction fetch
9. Machine Check, Internal Instruction Memory Error
10. Instruction Error
    a. Illegal instruction exception
    b. Illegal instruction sequence exception
11. Privilege Violation
    a. Access to 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
12. Memory error on XY Access
    a. Illegal AGU memory target address
    b. Unaligned AGU address spanning boundary
    c. 64-bit AGU destination address is not 32 bit aligned
13. Extension Instruction Exception - requested by extension instruction

14. Misaligned data memory access
15. TLB fault on data access
    a. Machine check, memory error on DTLB access
    b. DTLB miss
16. Protection Violation
    a. Protection Violation on data access (MMU, MPU, Stack Checking, or secure)
    b. Non-sequential execute target address MPU violations (overlap region, illegal secure to normal transition)
17. Memory error on data access:
    a. Data access spanning multiple data memory targets
    b. Cached LD/ST attempted on a peripheral interface
18. Machine check, internal data memory error
19. Divide by zero exception when STATUS32[DZ] = 1
20. Trap or software interrupt (TRAP_S, SWI or SWI_S instructions)
21. Bus error on data access to external data memory (or ICCM)
22. Actionpoint hit on Auxiliary register or Memory-access — an imprecise exception
23. Machine check - uncorrectable ECC or parity error in vector memory
24. Misaligned vector memory access
25. Vector stack pointer checking protection violation

The raising of Actionpoint exceptions on memory accesses or auxiliary registers accesses may be delayed beyond the point at which the triggering instruction completes, possibly allowing several subsequent instructions to be completed before the Actionpoint exception is taken. For further details on implementation-specific Actionpoint delays, see the related *ARCv3-based processor Databook*.

Table 7-3 describes the exception vectors, exception cause codes, and exception parameters.

7.4.3.1 APEX Exception Priority Levels

The APEX exceptions are raised in the following categories:

1. APEX Extension Group – User mode access is disabled by XPU
2. APEX Extension Group – Access to privileged feature

Within these APEX exceptions, the priority list for APEX events is as follows:

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

6. An extension core register is used as the second source register (the C operand register)
7. Extension auxiliary register access

7.4.3.2 Exception Vectors and Cause Codes

Table 7-3 indicates the ECR values when an exception is raised. If simultaneous violations occur, the Parameter field in ECR will have one bit set for each of those exceptions.

Table 7-4 on page 209 provides a summary of the values placed in the EFA and ERET registers for different exception types.

Table 7-3 Exception Vectors and Cause Codes

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
Reset	Reset	0x00	0x00	0x00	0x00	0x000000
Bus Error from Instruction Memory	Memory Error	0x08	0x01	0x00	0x0u	0x01000u
Bus Error from Data Memory	Memory Error	0x08	0x01	0x10	0x0u	0x01100u(b)
Bus Error from Data Memory during a Hardware table walk	Memory Error	0x08	0x01	0x41	0x0u	0x01410u0
Instruction Fetch spanning multiple instruction memory targets	Memory Error	0x08	0x01	0x02	0x00	0x010200
Instruction Fetch spanning multiple MPU regions	Memory Error	0x08	0x01	0x03	0x00	0x010300
Data access spanning multiple data memory targets	Memory Error	0x08	0x01	0x12	0x00	0x011200
Cached LD/ST instruction is attempted on a peripheral interface	Memory Error	0x08	0x01	0x15	0x00	0x011500
Memory error on Data Access	Memory Error	0x08	0x01	0x16	0x00	0x011600
Illegal Instruction	Instruction Error	0x10	0x02	0x00	0x00	0x020000
Illegal Instruction Sequence	Instruction Error	0x10	0x02	0x01	0x00	0x020100
Double Fault	EV_MachineCheck	0x18	0x03	0x00	0x00	0x030000

Table 7-3 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
Fatal Cache Error	EV_MachineCheck	0x18	0x03	0x03	0x00	0x030300
Internal Memory Error on Instruction Fetch	EV_MachineCheck	0x18	0x03	0x04	0xrr	0x0304rr(e)
Internal Memory Error on Data Access	EV_MachineCheck	0x18	0x03	0x05	0xrr	0x0305rr(e)
Illegal Overlapping MPU Entries	EV_MachineCheck	0x18	0x03	0x06	0x00	0x030600
Illegal Overlapping MPU entries (jump and branch target)	EV_MachineCheck	0x18	0x03	0x06	0x01	0x030601
Machine check - uncorrectable ECC or parity error in vector memory	EV_MachineCheck	0x18	0x03	0x80	0x00	0x038000
Translation fault exception on Instruction Fetch	EV_IMMUFault	0x20	0x04	0x00	0x00	0x040000
Translation fault exception on Invalid Instruction Address	EV_IMMUFault	0x20	0x04	0x08	0x00	0x040800
Access flag exception on Instruction Fetch	EV_IMMUFault	0x20	0x04	0x10	0x00	0x041000
Illegal Instruction Fetch ICCM translation	EV_IMMUFault	0x20	0x04	0x20	0x00	0x042000
Translation fault exception on Data Memory Read	EV_DMMUFault	0x28	0x05	0x01	0x00	0x050100
Translation fault exception on Data Memory Write	EV_DMMUFault	0x28	0x05	0x02	0x00	0x050200
Translation fault exception on Data Memory Read-Modify-Write (EX/AMOs)	EV_DMMUFault	0x28	0x05	0x03	0x00	0x050300
Translation fault exception on Invalid Data Address	EV_DMMUFault	0x28	0x05	0x08	0x00	0x050800

Table 7-3 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
Access flag exception on Data Access	EV_DMMUFault	0x28	0x05	0x10	0x00	0x051000
Illegal Data Access ICCM translation	EV_DMMUFault	0x28	0x05	0x20	0x00	0x052000
Illegal Data Access DCCM translation	EV_DMMUFault	0x28	0x05	0x30	0x00	0x053000
Instruction Fetch Protection Violation in MPU	EV_ProtV	0x30	0x06	0x00	0x04	0x060004
Instruction Fetch Protection Violation in MMU(d)	EV_ProtV	0x30	0x06	0x00	0x08	0x060008
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in stack checking scheme (parameter code 0x02)	EV_ProtV	0x30	0x06	0x01	0x02	0x060102
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in MPU (parameter code 0x04)	EV_ProtV	0x30	0x06	0x01	0x04	0x060104
Memory Read (LD, POP, LEAVE, interrupt exit, LLOCK) protection violation in MMU (parameter code 0x08)(d)	EV_ProtV	0x30	0x06	0x01	0x08	0x060108
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in stack checking scheme (parameter code 0x02)	EV_ProtV	0x30	0x06	0x02	0x02	0x060202
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in MPU (parameter code 0x04)	EV_ProtV	0x30	0x06	0x02	0x04	0x060204

Table 7-3 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
Memory Write (ST, PUSH, ENTER, interrupt entry, SCOND) protection violation in MMU (parameter code 0x08)(d)	EV_ProtV	0x30	0x06	0x02	0x08	0x060208
Memory Write Protection Violation from NVM	EV_ProtV	0x30	0x06	0x02	0x10	0x060210
Memory Read-Modify-Write (EX) protection violation in stack checking protection scheme (parameter code 0x02)	EV_ProtV	0x30	0x06	0x03	0x02	0x060302
Memory Read-Modify-Write (EX) protection violation in MPU scheme (parameter code 0x04)	EV_ProtV	0x30	0x06	0x03	0x04	0x060304
Memory Read-Modify-Write (EX) protection violation in MMU scheme (parameter code 0x08)(d)	EV_ProtV	0x30	0x06	0x03	0x08	0x060308
Memory Read-Modify-Write Protection Violation from NVM	EV_ProtV	0x30	0x06	0x03	0x10	0x060310
Action Point Hit, Instruction Fetch	EV_PrivilegeV	0x38	0x07	0x02	0xnn	0x0702nn
Privilege Violation	EV_PrivilegeV	0x38	0x07	0x00	0x00	0x070000
Disabled Extension	EV_PrivilegeV	0x38	0x07	0x01	0xnn	0x0701nn
Action Point Hit, Memory or Register	EV_PrivilegeV	0x38	0x07	0x02	0xnn	0x0702nn
Software Interrupt	EV_SWI	0x40	0x08	0x00	0xnn	0x0800nn
Trap	EV_Trap	0x48	0x09	0x00	0xnn	0x0900nn
Extension Instruction Exception	EV_Extension	0x50	0x0A	mm	0xnn	0x0Ammnn(c)
Invalid Operation, Floating-point extension exceptions	EV_Extension	0x50	0x0A	0x00	0x01	0x0A0001

Table 7-3 Exception Vectors and Cause Codes (Continued)

Exception	Vector Name	Vector Offset	Exception Cause Code Fields			ECR Value
			Vector Number	Cause Code	Para-meter	
Divide by Zero, Floating-point extension exceptions	EV_Extension	0x50	0x0A	0x00	0x02	0x0A0002
Divide by zero, Integer exception	EV_DivZero	0x58	0x0B	0x00	0x00	0x0B0000
Data cache consistency error	EV_DCError	0x60	0x0C	0x00	0x00	0x0C0000
Misaligned data access	EV_Misaligned	0x68	0x0D	0x00	0x00	0x0D0000
Misaligned data access: a LD or ST using an unaligned AGU based access straddles a modulo wrap boundary	EV_Misaligned	0x68	0x0D	0x10	0x00	0x0D1000
Reserved	-	0x78	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 216.

(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

nn = 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.

Table 7-4 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)	Physical address of the cache line	Imprecise exception, implementation-dependent
Memory access ECC or parity error on data-cache tag RAM, DCCM	Data-access address	PC address

7.4.3.3 Reset

A Reset is an external reset signal that causes the ARCv3-based processor to perform a *hard* Reset. Upon Reset, various internal states of the ARCv3-based processor are preset to their initial values, that is:

1. Interrupts are disabled.
2. The status register flags are cleared.
3. The scoreboard unit is cleared.
4. 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. On reset, the vector table base address is set by configuration parameter INTVBASE_PRESET (x1024) and reflected as a read-only value in build register Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD. The core registers are not initialized except loop count (which is cleared). The Reset value of vector base register determines the Reset vector address. This reset value is configured at build time through the INTVBASE_PRESET parameter. 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 ARCv3-based processor. For information about how to do a soft reset, see "Pass Control to an Handler or Perform a Soft Reset" on page 200.

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

7.4.3.4 Machine Check, Double Fault

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

7.4.3.5 Machine Check, Fatal TLB Error

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

7.4.3.6 Protection Violation, Instruction Fetch

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

7.4.3.7 Memory Error on Instruction Fetch

7.4.3.7.1 Bus Error Accessing External Instruction Memory

A bus error exception is raised when an error occurs during a memory access that occurs externally to the unit from which it originated. For example, if an error occurs during an instruction fetch that does not access ICCM or Instruction cache, 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.

7.4.3.7.2 Memory Error Exception, Access Spanning Multiple Memory Targets

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

- ICCM
- External memory

Note The DCCM and the peripheral Memory are not visible to the instruction fetch unit.

7.4.3.8 Memory Error Exception, Access Spanning Multiple MPU Regions

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

The MPU crossing can be legal or illegal. An unaligned instruction fetch access crossing an MPU boundary is considered illegal when:

- Core is operating in user-mode and the IC (code cacheability) or UE (user-mode execution permission) are different in both MPU regions
- Core is operating in kernel-mode and the IC (code cacheability) or KE (kernel-mode execution permission) are different in both MPU regions
- An unaligned instruction causing an illegal MPU crossing raises a memory error exception (vector number: 0x01, cause code: 0x03, parameter: 0x00).

An unaligned data read access crossing an MPU boundary is considered illegal when:

- Core is operating in user-mode and the UR (user-mode read permission), IDX (memory attribute index), or SH (shareability attribute) are different in both MPU regions
- Core is operating in kernel-mode and the KR (kernel-mode read permission), IDX (memory attribute index) or SH (shareability attribute) are different in both MPU regions

An unaligned data write access crossing an MPU boundary is considered illegal when:

- Core is operating in user-mode and the UW (user-mode write permission), IDX (memory attribute index) or SH (shareability attribute) are different in both MPU regions
- Core is operating in kernel-mode and the KW (kernel-mode write permission), IDX (memory attribute index) or SH (shareability attribute) are different in both MPU regions

An unaligned data access causing an illegal MPU crossing raises a memory error exception (vector number: 0x01, cause code: 0x12, parameter: 0x00).

7.4.3.9 Privilege Violation, Action-Point Hit Instruction Fetch

Action point hit triggered by instruction fetch. The parameter field (nn) gives the number of the action point that triggered the exception.

- 0x00 = AP0
- 0x01 = AP1
- 0x02 = AP2
- 0x03 = AP3
- 0x04 = AP4
- 0x05 = AP5
- 0x06 = AP6
- 0x07 = AP7

7.4.3.10 Machine Check, Internal Instruction Memory Error

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

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

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

7.4.3.11 Instruction Error

7.4.3.11.1 Illegal Instruction

If an invalid instruction is fetched that the ARCv3-based processor cannot execute, an Illegal Instruction exception is caused. Any use of an unimplemented instruction, condition code, core register, or auxiliary register raises an Illegal Instruction exception. This exception is also raised if a jump with a delay slot specifies long-immediate data as the target address operand.

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

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

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

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

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

Any attempt to write an incorrect value to an MMU command register. This exception sets the Parameter field of the Exception Cause Register (ECR) to the value 0x01 to indicate an attempt to write an invalid command to an MMU control register.

7.4.3.11.2 Illegal Instruction Sequence

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

- When any of the following instructions are attempted in the delay slot of a taken jump or branch:
  - Another jump or branch instruction (Bcc, BLcc, Jcc, JLcc, and so on)

- A return from interrupt instruction (RTIE)
- Any instruction with long-immediate data as a source operand
- An ENTER_S or LEAVE_S instruction

- When a LIMM is present in the delay-slot instruction for any of the branch or jump instructions that set BLINK (BL.D, BLcc.D, JL.D, or JLcc.D)

7.4.3.12 Privilege Violation

7.4.3.12.1 Privilege Violation, Kernel Only Access

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

7.4.3.12.2 Privilege Violation, Disabled Extension

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

7.4.3.13 Extension Instruction Exception

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

The following are supplied by the extension instruction:

- mm = subcode
- nn = parameter

Floating-point Extension

mm = 0 — indicates floating-point extension exceptions

nn—indicates specific exception(s) raised

nn	=	1 —Invalid Operation floating-point exception; the double-precision floating-point accelerator instructions (DADD*x*, DSUB*x*, DMUL*x*, DEXCL*x)* 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.

7.4.3.14 Misaligned Data Access

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

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 will contain the address that triggered the exception.

7.4.3.15 Translation fault exception on Instruction Fetch

A translation fault on Instruction fetch exception is triggered when a page-table-walk on behalf of a L2 TLB finds an invalid or reserved descriptor (bits [1:0]). The lookup level where the fault occurred is reported at the MMU_FAULT_STATUS auxiliary register.

7.4.3.16 Translation fault exception on Data Access

A translation fault on Data access exception is triggered when a page-table-walk on behalf of a L2 TLB miss finds an invalid or reserved descriptor (bits [1:0]). The lookup level where the fault occurred is reported at the MMU_FAULT_STATUS auxiliary register.

7.4.3.17 Translation fault exception on Invalid Address

This translation fault is triggered the input address for translation is outside of the range specified by T0SZ (RTP0 region) or T1SZ (RTP1 region). The lookup level for an invalid address fault is reported as level 0 in the MMU_FAULT_STATUS auxiliary register

7.4.3.18 Access Flag Exception On Instruction Fetch Or Data Access

This exception is triggered when the hardware page table walk accesses a page descriptor with the access flag set to 0. The lookup level where the fault occurred is reported at the MMU_FAULT_STATUS auxiliary register.

7.4.3.19 Illegal Instruction Fetch ICCM Translation

This exception is triggered when an instruction fetch access to the ICCM access violates the CCM-MMU rules

7.4.3.20 Illegal Data Access ICCM Translation

This exception is triggered when a data fetch (DMP) from ICCM violates the CCM-MMU rules

7.4.3.21 Illegal Data Access DCCM Translation

This exception is triggered when a data fetch (DMP) from DCCM violates the CCM-MMU rules

7.4.3.22 Bus Error From Data Memory During A Hardware Table Walk

This exception is triggered when the hardware page table walk encounters a bus error exception.

7.4.3.23 Data Access Spanning Multiple Data Memory Targets

This exception is triggered when an unaligned load/store instruction crosses a page boundary and the pages have different memory index or different shareability attributes

Page table entries causing a translation fault or access flag exception are never cached in the TLB. As such, these exception handlers do not have to perform TLB maintenance instructions to remove such entries from the TLB.

7.4.3.24 Machine Check, Instruction Fetch Memory Error

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

7.4.3.25 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).

7.4.3.26 Memory Error on Data Access

7.4.3.26.1 Cached Access to a Peripheral Interface

A memory error exception is raised when a cached data read or write is attempted on a peripheral interface.

7.4.3.26.2 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.

7.4.3.27 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.

7.4.3.28 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.

7.4.3.29 Trap or Software Interrupt

7.4.3.29.1 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.

7.4.3.29.2 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.

7.4.3.29.3 Bus Error on Data Access to External Data Memory

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

7.4.3.30 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.

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

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

7.4.3.32 Misaligned Vector Memory Access

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

7.4.3.33 Vector Stack Pointer Checking Protection Violation

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

7.4.4 Exception Detection

Precise 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. 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.

7.4.5 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.

7.4.6 Exception Entry

This section describes actions taken when an exception is taken. All addresses in this description are the logical addresses determined by the program.

1. All speculative instructions that are in a partial state of execution are dismissed.
   - No state changes arising from a dismissed instruction are committed
   - Likewise, all APEX state changes associated with extension core registers or condition codes must also be prevented by APEX extensions when an APEX 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 7-3.
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.

7.4.6.1 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.

7.4.7 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 7-5.

In Table 7-5, 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 7-5 Exception and Interrupt Exit Modes

U	AE	AUX_IRQ_ACT	FIRQ_OPTION	Current Mode	RTIE Response	Link Registers Used
0	0	0000_0000_0000_0000	N/A	Kernel	Exception Exit	ERET, ERSTATUS, ERBTA
0	0	xxxx_xxxx_xxxx_xxx1	1	ISR P0	Interrupt Priority Level P0 Exit	ILINK, STATUS32_P0
0	0	xxxx_xxxx_xxxx_xxx1	0	ISR P0	Interrupt Priority Level P0 Exit	Restore from stack
0	0	xxxx_xxxx_xxxx_xx10	N/A	ISR P1	Interrupt Priority Level P1 Exit	Restore from stack
0	0	xxxx_xxxx_xxxx_x100	N/A	ISR P2	Interrupt Priority Level P2 Exit	Restore from stack
0	0	xxxx_xxxx_xxxx_1000	N/A	ISR P3	Interrupt Priority Level P3 Exit	Restore from stack
0	0	xxxx_xxxx_xxx1_0000	N/A	ISR P4	Interrupt Priority Level P4 Exit	Restore from stack
0	0	xxxx_xxxx_xx10_0000	N/A	ISR P5	Interrupt Priority Level P5 Exit	Restore from stack
0	0	xxxx_xxxx_x100_0000	N/A	ISR P6	Interrupt Priority Level P6 Exit	Restore from stack
0	0	xxxx_xxxx_1000_0000	N/A	ISR P7	Interrupt Priority Level P7 Exit	Restore from stack
0	0	xxxx_xxx1_0000_0000	N/A	ISR P8	Interrupt Priority Level P8 Exit	Restore from stack
0	0	xxxx_xx10_0000_0000	N/A	ISR P9	Interrupt Priority Level P9 Exit	Restore from stack
0	0	xxxx_x100_0000_0000	N/A	ISR P10	Interrupt Priority Level P10 Exit	Restore from stack
0	0	xxxx_1000_0000_0000	N/A	ISR P11	Interrupt Priority Level P11 Exit	Restore from stack

Table 7-5 Exception and Interrupt Exit Modes (Continued)

U	AE	AUX_IRQ_ACT	FIRQ_OPTION	Current Mode	RTIE Response	Link Registers Used
0	0	xxx1_0000_0000_0000	N/A	ISR P12	Interrupt Priority Level P12 Exit	Restore from stack
0	0	xx10_0000_0000_0000	N/A	ISR P13	Interrupt Priority Level P13 Exit	Restore from stack
0	0	x100_0000_0000_0000	N/A	ISR P14	Interrupt Priority Level P14 Exit	Restore from stack
0	0	1000_0000_0000_0000	N/A	ISR P15	Interrupt Priority Level P15 Exit	Restore from stack
0	1	xxxx_xxxx_xxxx_xxx	N/A	Exception	Exception Exit	ERET, ERSTATUS, ERBTA
1	-	xxxx_xxxx_xxxx_xxx	N/A	User	Privilege Violation	N/A

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

7.4.8 Exceptions and Delay Slots

Exceptions can be raised by instructions in the delay slots of branches.

Example 7-9 Exception in a Delay Slot

```
J.D [blink]    ; Branch/Jump Instruction
LD  fp,[sp,24] ;
:              ;
MOV r0,0       ; Target of the branch/jump
```

The ARCv3 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.

However, a delay-slot instruction is not permitted to have a long-immediate operand. It is illegal for an instruction appearing in the delay-slot of a branch or jump instruction to have a long-immediate data operand (LIMM). The processor will ignore any such LIMM operand specified by a delay-slot instruction, and will consequently determine the size of the delay-slot instruction based solely on its major opcode. If the preceding branch or jump instruction sets the BLINK register (e.g. JL.D or BL.D) then the BLINK value defined by that instruction will be the address of its delay-slot instruction plus the size of its delay-slot instruction, but always excluding any illegal LIMM operand.

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 will always observe 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 hence 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 will complete 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 in order to resume execution from the parent branch."

7.4.9 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)
  - ERET + emulated_instruction_size for normal, linear execution (see Exception Return Address, ERET)

7.4.10 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.

8 Instruction Set Summary

This chapter provides an overview of the ARCv3 ISA. It presents an overview of how the ARCv3 ISA is encoded, explains the assembler syntax conventions used by the remainder of this document, and presents a summary of the instructions found in each functional instruction grouping. Information the encoding of each instruction can be found in Chapter 9, "32-bit Instruction Formats Reference" and Chapter 10, "16-bit Instruction Formats Reference", and full details of all aspects of each individual instruction can be found in Chapter 11, "Instruction Set Details".

Among the defining characteristics of ARC processors is their configurability and extensibility. The ARCv3 instruction set is therefore highly configurable. All ARCv3 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.

8.1 Instruction Set Encoding

The ARCv3 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.

8.1.1 Top-level Instruction Formats

Table 8-1 lists the 32 top-level instruction formats of the ARCv3 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 8-1 Top-level Formats of the 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 BRccL 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_APEX	User 32-bit extension instructions	op a,b,c
F16_MV_HREG	Move/Load with h-register	MOV_S g,h / LD_S R0-3,[h,u5]
F16_LD_ADD_SUB	Load/Add/Sub compact	LD_S.AS a,[b,c] / SUB_S a,b,c / ADD_S R0-1,b,u6
F16_LD_ST_JLI	Load/Store gp-relative Load-indexed in ARC64	LD_ST_R01 JLI_S_U10
F16_LD_ADD_RR	Load/Add register-register	LD_S / LDB_S / LDH_S / ADD_S a,b,c
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]
F16_SH_SUB_BIT	Shift/Subtract/bit ops	op_S b,b,u5
F16_SP_OP64	SP-based Load/Store/Add/Subtract Function prolog, epilog; available in ARC64	LD_S, LDB_S, ST_S b, STB_S b, ADDL_S, SUBL_S, POPL_S, POPDL_S, PUSHL_S, PUSHDL_S, LEAVE_S, ENTER_S

Table 8-1 Top-level Formats of the Instruction Set (Continued)

Format	Description	Examples
F16_GP_OP64	GP-based Load/Add; available in ARC64	LD_S / LDH_S / LDB_S / ADDL_S
F16_PCL_LD	PCL-based Load	LD_S b,[PCL,u10]
F16_MV_IMM	Move immediate	MOV_S b,u8
F32_FP_OPS	Available in ARC64	
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

8.1.2 Instruction Set Profiles

The ARCv3 instruction set includes the concept of an **instruction set profile**. Each profile allows a small number of top-level formats to encode profile-specific sets of extension instructions that are designed for the application domains of that profile. This acknowledges the fact that a common set of instructions are applicable across all domain, although a unique use of every encoding is too restrictive for all potential use-cases.

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

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

The **64-bit Profile** allocates one major opcode to support 64-bit operations, as well as permitting the majority of 16-bit encodings for enhanced code density. This profile is designed to support 64-bit server and host workloads, where wider internal datapaths and 64-bit virtual addresses are required.

8.1.3 Assignment of Instruction Formats to Major Opcodes

Table 8-2 shows the assignment of the top-level instruction formats to the major opcodes of the ARCv3 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 8-2 Assignment of Top-level Instruction Formats to Major Opcodes in each Instruction Set Profile

Major opcode	Minor opcode	ARCv3 Profile-specific formats			
		Baseline	Compact	Vector DSP	ARC64
0x00	-	F32_BR0			
0x01	-	F32_BR1			
0x02	-	F32_LD_OFFSET			
0x03	-	F32_ST_OFFSET			
0x04	-	F32_GEN4			
0x05	-	F32_EXT5			
0x06	-	Illegal			
0x07	-	F32_APEX			
0x08	inst[2] = 0	F16_COMPACT0			
	inst[2] = 1				F16_MOVL
0x09	-		F16_LD_ADD_SUB		
0x0A	inst[3] = 0		LD_ST_R01		
	inst[3] = 1				JLI_S_U10
0x0B	-				F32_GEN_OP64
0x0C	-	F16_LD_ADD_RR			
0x0D	-	F32_FP_MEM			
0x0E	-	F16_OP_HREG			
0x0F	-	F16_GEN_OP			
0x10	-	F16_LD_WORD			
0x11	-		F16_LD_BYTE	F32_VEC1	F16_LD_BYTE
0x12	-		F16_LD_HALF	F32_VEC2	F16_LD_HALF
0x13	-		F16_LDX_HALF	F32_VEC3	F16_LDX_HALF
0x14	-	F16_ST_WORD			
0x15	-		F16_ST_BYTE	F32_VEC4	F16_ST_BYTE
0x16	-		F16_ST_HALF	F32_VEC5	F16_ST_HALF
0x17	-	F16_SH_SUB_BIT			
0x18	-				F16_SP_OP64
0x19	-				F16_GP_OP64
0x1A	-	F16_PCL_LD			
0x1B	-	F16_MV_IMM			

Table 8-2 Assignment of Top-level Instruction Formats to Major Opcodes in each Instruction Set Profile

Major opcode	Minor opcode	ARCv3 Profile-specific formats			
		Baseline	Compact	Vector DSP	ARC64
0x1C	-	F32_FP_OPS (floating-point option for all profiles)			
0x1D	-	F16_BCC_REG			
0x1E	-	F16_BCC			
0x1F	-		F16_BL	F32_VEC6	F16_BL

8.1.4 32-bit Instruction Formats

Table 8-3 summarizes the field layout of the 32-bit formats in the ARCv3 ISA, arranged hierarchically by (a) the top-level format, (b) the number of operands in the format, and (c) the operand sub-formats.

Table 8-3 Summary of 32-bit instruction formats

32-bit Format Hierarchy (a)	(b)	(c)	Major Opcode	Layout of Instruction Formats (31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0)
F32_BR0		COND	0x0	S[10:1] 0 S[20:11] N Q[4:0]
		UCOND_FAR		S[10:1] 1 S[20:11] N R S[24:21]
F32_BR1	BL	COND	0x1	S[10:2] 0 0 S[20;11] N Q[4:0]
		UCOND_FAR		S[10:2] 1 0 S[20;11] N R S[24:21]
	BCC	REG_REG		B[2:0] S[7:1] 1 S8 B[5:3] C[5:0] N 0 i[3:0]
		REG_U6		B[2:0] S[7:1] 1 S8 B[5:3] U[5:0] N 1 i[3:0]
F32_LD_OFFSET			0x2	B[2:0] S[7:0] S8 B[5:3] D aa ZZ X A[5:0]
F32_ST_OFFSET		ST_REG	0x3	B[2:0] S[7:0] S8 B[5:3] C[5:0] D aa ZZ 0
		ST_W6		B[2:0] S[7:0] S8 B[5:3] W[5:0] D aa ZZ 1
F32_GEN4	DOP	REG_REG	0x4	B[2:0] 0 0 i[5:0] except when (i[5:3] == 110) or (i[5:0] == 101111) F B[5:3] C[5:0] A[5:0]
		REG_U6		B[2:0] 0 1 i[5:0] except when (i[5:3] == 110) or (i[5:0] == 101111) F B[5:3] U[5:0] A[5:0]
		REG_S12		B[2:0] 1 0 i[5:0] except when (i[5:3] == 110) or (i[5:0] == 101111) F B[5:3] S[5:0] S[11:6]
		COND_REG		B[2:0] 1 1 i[5:0] except when (i[5:3] == 110) or (i[5:0] == 101111) F B[5:3] C[5:0] 0 Q[4:0]
		COND_U6		B[2:0] 1 1 i[5:0] except when (i[5:3] == 110) or (i[5:0] == 101111) F B[5:3] U[5:0] 1 Q[4:0]
		LD_REG		B[2:0] a a 1 1 0 ZZ X D B[5:3] C[5:0] A[5:0]
	SOP	REG_REG	0x4	B[2:0] 0 0 1 0 1 1 1 1 F B[5:3] C[5:0] i[5:0]
		REG_U6		B[2:0] 0 1 1 0 1 1 1 1 F B[5:3] U[5:0] i[5:0]
	ZOP	REG_REG	0x4	i[2:0] 0 0 1 0 1 1 1 1 F i[5:3] C[5:0] 1 1 1 1 1 1
		REG_U6		i[2:0] 0 1 1 0 1 1 1 1 F i[5:3] U[5:0] 1 1 1 1 1 1
F32_EXT5, F32_APEX F32_GEN_OP64	DOP	REG_REG	0x5, 0x6, 0x7 0xB	B[2:0] 0 0 i[5:0] except when (i[5:0] == 101111) F B[5:3] C[5:0] A[5:0]
		REG_U6		B[2:0] 0 1 i[5:0] except when (i[5:0] == 101111) F B[5:3] U[5:0] A[5:0]
		REG_S12		B[2:0] 1 0 i[5:0] except when (i[5:0] == 101111) F B[5:3] S[5:0] S[11:6]
		COND_REG		B[2:0] 1 1 i[5:0] except when (i[5:0] == 101111) F B[5:3] C[5:0] 0 Q[4:0]
		COND_U6		B[2:0] 1 1 i[5:0] except when (i[5:0] == 101111) F B[5:3] U[5:0] 1 Q[4:0]
	SOP	REG_REG	0x5, 0x6, 0x7 0xB	B[2:0] 0 0 1 0 1 1 1 1 F B[5:3] C[5:0] i[5:0] (except 111111)
		REG_U6		B[2:0] 0 1 1 0 1 1 1 1 F B[5:3] U[5:0] i[5:0] (except 111111)
	ZOP	REG_REG	0x5, 0x6, 0x7 0xB	i[2:0] 0 0 1 0 1 1 1 1 F i[5:3] C[5:0] 1 1 1 1 1 1
		REG_U6		i[2:0] 0 1 1 0 1 1 1 1 F i[5:3] U[5:0] 1 1 1 1 1 1

Table 8-4 The F32_FP_OPS Instruction Formats

Format	Sub-format		[31:27]	[26:24]	23	22	21	20	19	18	17	16	15	14	13	12	11	[10:6]	5	4	3	2	1	0
F32_FP_OPS	FP_TOP	>	0x1C	fs2[2:0]	fs3					topf[3:1]			P		fs2[4:3]		1	fd	topf[0]	fs1				
	FP_DOP	>			0	0	-	dopf									0	fd	1	fs1				
	FP_CVF2F	>			0	1	1	0	-	-	cvtf		-				0	fd	1	u4	u3	1	0	u0
	FP_RND	>			0	1	1	0	-	-							0	fd	1	0	u3	1	1	0
	FP_CVF2I	>			0	1	1	0	-	-							0	rd	1	0	u3	0	u1	1
	FMVF2I	=			0	1	1	0	-	-							0	rd	1	1	0	0	0	1
	FP_SOP	>		-	0	1	0	0	0	-	sopf		P		-		0	fd	1	fs1				
	FP_COP	>		Q[2:0]	0	1	0	0	1	-	copf		P		Q[4:3]		0	fd	1	fs1				
	FP_ZOP	>		-	0	1	1	1	-	-	zopf		-		-		0	-	1	-				
	FP_VMVI	>		u[2:0]	0	1	0	1	-	-	vmvf		P		u[4:3]		0	fd	1	fs1				
	FP_VMVR	>		B[2:0]	1	0	-	-	-	-			P		B[4:3]		0	fd	1	fs1				
	FP_CVI2F	>			1	1	1	0	-	-	cvtf		-		B[4:3]		0	fd	1	0	u3	0	u1	0
	FMVI2F	=			1	1	1	0	-	-			-				0	fd	1	1	0	0	0	0
	RESERVED																0		0					

Table 8-5 The F32_FP_MEM Instruction Formats

Format	Sub-format		[31:27]	[26:24]	23	22	21	20	19	18	17	16	15	14	13	12	11	[10:6]	5	4	3	2	1	0
F32_FP_MEM	FP_LOAD	>	0x0D	B[2:0]	S[7:0]								S8	B[5:3]			0	fd	d	a a		ZZ		0
	FP_STORE	>		B[2:0]									S8	B[5:3]			0	fs1	d	a a		ZZ		1

8.1.5 16-bit Instruction Formats

Table 8-6 summarizes the 16-bit compact instruction formats defined by the ARCv3 instruction set.

Table 8-6 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	F16_COMPACT0	>	0x8	g[2:0]						g[4:3]		**0**	h[4:3]						
	F16_MOVL	>		g[2:0]			h[2:0]			g[4:3]		**1**	h[4:3]						
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_JLI	LD_ST_R01	>	0xA	S[10:5]						i	**0**	S[4:2]							
	JLI_S_U10	>		U[9:3]							**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_OP_HREG	HREG_REG	>	0xE	b[2:0]			h[2:0]			i[1:0]		**0**	h[4:3]						
	HREG_S3	>		S[2:0]								**1**							

Table 8-6 Summary of 16-bit Instruction Formats (Continued)

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_OP64		>	0x18	b[2:0]	i[2:0]	U[4:0]
F16_GP_OP64		>	0x19	i[1:0]	S[8:0]	
F16_PCL_LD		>	0x1A	b[2:0]	U[7:0]	
F16_MV_IMM		>	0x1B			
F32_FP_OPS		>	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]		

8.1.6 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 329. Table 8-19 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 8-7 and Table 8-8 lists the notation used for the encoding.

Table 8-7 Key for 32-bit Addressing Modes and Encoding Conventions

Encoding Character	Encoding Field	Syntax
I	I[4:0]	instruction major opcode
i	i[n:0]	instruction sub opcode
A	A[5:0]	destination register
b	B[2:0]	lower bits source/destination register
B	B[5:3]	upper bits source/destination register
C	C[5:0]	source/destination register
Q	Q[4:0]	condition code
u	U[*n*:0]	unsigned immediate (number is bitfield size)
s	S[*n*:0]	lower bits signed immediate (number is bitfield size)
S	S[*m*:*n*+1]	upper bits signed immediate (number is bitfield size)
T	S[24:21]	upper bits signed immediate (branch unconditionally far)
w	W[*5*:0]	signed immediate 6-bit store data value
P	P[1:0]	operand format
M	M	conditional instruction operand mode
N	N	<.d> delay slot mode
F	F	Flag Setting
R	R	Reserved
D	Di	<.di> direct data cache bypass
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 8-8 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)

Table 8-8 Key for 16-bit Addressing Modes and Encoding Conventions (Continued)

Encoding Character	Encoding Field	Syntax
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-r31excluding r29 and r30)
u	u[*n*:0]	unsigned immediate (number is bitfield size)
s	s[*n*:0]	signed immediate (number is bitfield size)

8.1.7 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.

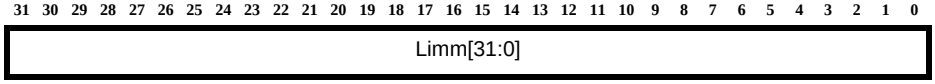
When a LIMM is required as an operand in a 64-bit context, the ARC64 ISA provides two variants of the LIMM; one that zero-extends the LIMM to 64 bits, and another that sign-extends the LIMM to 64 bits. The zero-extended version is specified using the existing LIMM indicator of r62. The new sign-extended version is specified using a new LIMM indicator of r60.

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 the section on "Instruction Layout in Memory" on page 106.

Figure 8-1 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)

8.1.8 Condition Code Tests

Table 8-9 lists the codes used for condition code tests.

Table 8-9 Condition Codes

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

8.1.8.1 Condition Code Tests for Floating-Point Instructions

Table 8-10 lists the mapping from the ARCv3 condition codes to the IEEE-754 standard condition codes:

Table 8-10 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
	PNZ	Positive non-zero	/N and /Z	0x0F

Note PNZ does not have an inverse condition.

8.1.9 Load or Store Instructions

The load and store instructions have the following capabilities:

- Acquire/release semantics for release consistency and relaxed memory ordering (optional)
- 64-bit load/store operations in ARC64-based processors
- Optional 128-bit load/store operations in ARC64-based processors

Load and store instructions that are new in ARCv3 are encoded without altering the encodings of preexisting load/store instructions inherited from ARCv2.

8.1.9.1 Acquire or Release Semantics for Loads and Stores

To support C11/C++11 atomics requires the ability to specify load operations with acquire semantics and store operations with release semantics. These capabilities are provided for C/C++ scalar types; vector types do not need these capabilities. ARC64 supports scalar types corresponding to Byte (8b), Halfword(16b), Word (32b), and include Long-word (64b). The ARCv3 ISA also provide variants of each scalar sub-register load that sign-extends the loaded value to perform an integer promotion to full register size (that is, 64b for ARC64).

The ARCv3 ISA provides a variant of each scalar load operation (with and without sign-extension) that has acquire semantics. This is indicated in the load syntax by specifying an `.aq` extension. The `.aq` extension cannot be combined with any of the address modifier or uncached extensions.

The ARCv3 ISA provides a variant of each scalar store operation that has release semantics. This is indicated in the store syntax by specifying an `.rl` extension. The `.rl` extension cannot be combined with any of the address modifier or uncached extensions.

The `.aq` extension is not recognized as an extension for store operations, and `.rl` is not recognized as an extension for load operations.

Instructions that optionally support both acquire and release semantics shall use the syntax `.aq.rl` to indicate that both are selected.

The acquire/release semantics are not provided orthogonally with respect to addressing modes and cached/uncached operations, nor are they provided for store instructions using the w6 store-data operand.

Instructions with acquire and release semantics are supported only when the -atomic_option 3 is selected during core configuration.

8.1.9.2 Acquire/Release semantics for LLOCK and SCOND

The LLOCK.AQ and SCOND.>RL instructions provide acquire and release semantics to the LLOCK and SCOND instructions respectively. The LLOCK.AQ and SCON>RL instructions are encoded in the F32_GEN_OP format (major opcode 0x4), in the SOP sub-format. LLOCK.AQ uses opcode 0x14 and SCOND.RL uses opcode 0x15, both with the F bit set to 0. The instructions LLOCK, LLOCK.DI, SCOND, SCOND.DI are implemented without acquire or release semantics.

Acquire and release semantics are supported only when the -atomic_option core configuration parameter is set to 3.

8.1.9.3 Supporting 128-bit Loads and Stores

To support load or store of two 64-bit registers per cycle in function prolog/epilog and interrupt prolog/epilog sequences, 64-bit equivalents of the ARC32 LDD and STD that work on pairs of 64-bit registers are supported when the -m128_option core configuration option is enabled. ARC64then defines two new instructions: LDDL (load double long) and STDL (store double long). These instructions load or store a short vector of two 64-bit values between memory and two 64-bit registers, given an even-numbered register operand (similar to LDD and STD). The ISA supports LDDL and STDL through the new configuration switch `-m128_option`, which is set to `true` to enable support for double long-word loads and stores.

The ARC64 ISA has the capability to encode LDD and STD instructions, but specific products may choose to disallow the `-ll64_option`. This is the case for ARC HS6x processors.

The LDD, STD, LDDL, and STDL instructions do not support the capability to specify acquire/release semantics. This is because they are effectively vector load/store operations, and therefore are not required by C11/C++11 atomics.

8.1.9.4 Encoding Load Instructions in ARCv3

There were two major formats that encoded Load instructions in ARCv2, and in each case, there were four sub-fields used in ARCv2 to orthogonally define all variants of load instruction. Those fields were:

- AA – specified the addressing modifier mode (none, .AW, .AB, .AS)
- DI – specified whether the load is cacheable (default) or non-cacheable (.DI)
- X – specified whether the value is zero-extended (default) or sign-extended (.X)
- ZZ – specified the data size (byte, half, word, double-word)

There were some combinations of these field values that were not meaningful, and therefore represented illegal encodings. In ARCv3 these previously illegal encodings have been assigned to encode new variants of load instruction. However, all legacy ARCv2 load instructions are supported in ARCv3, and their encodings are unchanged. Hence, the ARCv3 load instructions are backwards binary compatible with ARCv2.

In general, the new ARCv3 load instructions do not support uncached variants. Software should therefore use MMU page attributes or MPU region attributes to control cacheability of memory accesses.

ARC64 supports a version of load-word that performs sign extension (LD.X) and a new load-long (LDL) instruction. Although these each support all addressing modes, they do not support uncached versions.

ARC64 supports 128-bit memory operations only when the `-m128_option` is configured. This adds the LDDL variant of the load instruction that are shown in Table 8-11.

The configurations of the ARCv3 ISA supports load instructions with acquire semantics only when the -atomic_option ISA configuration level is set to 3.

The encodings for each supported variant of ARCv3 load instructions are defined in terms of the X, ZZ, AA and DI fields within the load instruction formats, as shown in the Table 8-11. The positions of these fields, see Load Register with Offset Format [F32_LD_OFFSET] and Load Register-Register, 0x04, [0x30 – 0x37].

Table 8-11 Load Instruction Encodings

{X ZZ} Fields	{ AA DI } Fields							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 01	LDB	LDB.DI	LDB.AW	LDB.AW.DI	LDB.AB	LDB.AB.DI	LDDL	LDDL.AS
1 01	LDB.X	LDB.X.DI	LDB.X.AW	LDB.X.AW.DI	LDB.X.AB	LDB.X.AB.DI	LDDL.AW	LDDL.AB
0 10	LDH	LDH.DI	LDH.AW	LDH.AW.DI	LDH.AB	LDH.AB.DI	LDH.AS	LDH.AS.DI
1 10	LDH.X	LDH.X.DI	LDH.X.AW	LDH.X.AW.DI	LDH.X.AB	LDH.X.AB.DI	LDH.X.AS	LDH.X.AS.DI

Table 8-11 Load Instruction Encodings

{X ZZ} Fields	{ AA DI } Fields							
0 00	LD	LD.DI	LD.AW	LD.AW.DI	LD.AB	LD.AB.DI	LD.AS	LD.AS.DI
1 00	LD.X	LDL	LD.X.AW	LDL.AW	LD.X.AB	LDL.AB	LD.X.AS	LDL.AS
0 11	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
1 11	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

8.1.9.5 Encoding Store Instructions in ARCv3

There was one major format that encoded Store instructions in ARCv2, which supported four orthogonal options to define all the possible variants of each store instruction. Those options were:

- AA – specified the addressing modifier mode (none, .AW, .AB, .AS)
- DI – specified whether the store was cacheable (default) or non-cacheable (.DI)
- K – specified whether the store data value was provided by a 6-bit literal or a register
- ZZ – specified the data size (byte, half, word, double-word)

There were some combinations of these options that were not meaningful, and therefore represented illegal instructions. In ARCv3 these previously illegal encodings have been assigned to encode new variants of store instruction. However, all legacy ARCv2 store instructions are supported in ARCv3, and their encodings are unchanged. Hence, the ARCv3 store instructions are backwards binary compatible with ARCv2.

The new ARCv3 store instructions do not support uncached variants. Software should therefore use MMU page attributes or MPU region attributes to control cacheability of memory accesses.

ARC64 supports a store-long (STL) instruction. The four variants of STL are in Table 8-12. Although these each support all addressing modes, they do not have uncached variants.

ARC64 supports 128-bit memory operations when the -m128_option is configured. This adds four variants of the store double-long (STDL) instruction, which are shown in Table 8-12.The ARCv3 ISA supports store instructions with release semantics only when the -atomic_option ISA configuration level is set to 3.

Table 8-12 Store Instruction Encodings

{K ZZ} Fields	{ AA DI } Fields							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 01	STB c	STB.DI c	STB.AW c	STB.AW.DI c	STB.AB c	STB.AB.DI c	Illegal	Illegal
0 10	STH c	STH.DI c	STH.AW c	STH.AW.DI c	STH.AB c	STH.AB.DI c	STH.AS c	STH.AS.DI c
0 00	ST c	ST.DI c	ST.AW c	ST.AW.DI c	ST.AB c	ST.AB.DI c	ST.AS c	ST.AS.DI c
0 11	Illegal	STDL c	Illegal	STDL.AW c	Illegal	STDL.AB c	Illegal	STDL.AS c

Table 8-12 Store Instruction Encodings

{K ZZ} Fields	{ AA DI } Fields							
1 01	STB w6	STB.DI w6	STB.AW w6	STB.AW.DI w6	STB.AB w6	STB.AB.DI w6	Illegal	Illegal
1 10	STH w6	STDL w6	STH.AW w6	STDL.AW w6	STH.AB w6	STDL.AB w6	STH.AS w6	STDL.AS w6
1 00	ST w6	ST.DI w6	ST.AW w6	ST.AW.DI w6	ST.AB w6	ST.AB.DI w6	ST.AS w6	ST.AS.DI w6
1 11	STL c	STL w6	STL.AW c	STL.AW w6	STL.AB c	STL.AB w6	STL.AS c	STL.AS w6

8.1.9.6 Scaled Addressing Mode (.AS) for 64-bit and 128-bit Loads/Stores

When the `.AS` addressing mode is used with 64-bit or 128-bit loads and stores the address index is scaled by a factor of 8. This applies to LDL.AS, STL.AS, LDDL.AS and STDL.AS instructions. Note, the index is determined by the size of the scalar items being referenced; for LDDL and STDL the element size is 8 bytes, although these instructions each transfer two such elements.

8.1.10 Encoding Branch and Jump Delay Slot Modes

Table 8-13 lists the codes used for delay slot modes on Branch and Jump instructions.

Table 8-13 Delay Slot Modes

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

8.1.10.1 BRcc, BBIT0 and BBIT1 Instruction Semantics

The existing 32-bit BRcc and BBITn instructions are preserved in ARC64, but with the restriction that the static branch prediction option is no longer supported . The previous ARCv2 BRcc and BBITn instructions had a <.T> field (in bit position 3) that could be used to encode a static branch prediction hint. The ARC64 versions of BRcc and BBITn repurpose bit 3 to differentiate between 32-bit and 64-bit versions of BRcc/BBITn instructions (BRccL and BBITnL).

In addition to the 32-bit compare-and-branch operations (BREQ, BRNE, BRLT, BRGE, BRLO and BRHS) there is an orthogonal set of 64-bit compare-and-branch operations: BREQL, BRNEL, BRLTL, BRGEL, BRLOL and BRHSL. These perform equivalent tests, but using comparisons over the full 64-bit operands.In addition to the pre-existing 32-bit bit-test and branch operations (BBIT0 and BBIT1), there are orthogonal 64-bit bit-test and branch operations: BBIT0L and BBIT1L. The 32-bit BBITn instructions use only the lower 5 bits of the bit-selection field [c | u6] to select one of bits 0 through 31 of their b operand. In contrast, the 64-bit BBITnL instructions use the lower 6 bits of [c | u6] to select one of bits 0 through 63 of their b operand.

The compact encodings previously used for BREQ_S and BRNE_S are redefined in ARC64 as BREQL_S and BRNEL_S. This allows these instructions to be used in both 64-bit and 32-bit contexts. The 32-bit case

requires a little explanation. When comparing a 32-bit value against zero using BREQL_S and BRNEL_S, the b operand is a normally a 64-bit register. If the value in that register was produced by a 32-bit instruction, then the upper 32 bits will be zero. If the b operand would be a LIMM, then the branch condition can be evaluated statically and the compiler can either eliminate the branch or convert it to an unconditional branch. Therefore, a 64-bit comparison against zero will be influenced only by the lower 32 bits, and hence a 64-bit comparison works in both 64-bit and 32-bit contexts.

8.1.11 Semantics of Load / Store Address Write-back Modes

Table 8-14 lists the address write-back modes supported by Load and Store instructions, explaining for each one how the memory address is computed and what value (if any) is written back to the base address register.

Table 8-14 Address Write-Back Modes

Address Mode	Memory Address Used	Register Value Write-back
No write-back	Reg + offset	No write-back
.A or .AW	Reg + offset	Reg + offset Register updated pre memory transaction.
.AB	Reg	Reg + offset Register updated post memory transaction.
.AS Scaled, no write-back	Reg + (offset << scaling_shift) Note that using the scaled address mode with 8-bit data size (LDB.AS or STB.AS) has undefined behavior and must not be used. scaling_shift is based on the data size mode, ZZ. For more information, see Table 8-17 scaling_shift is defined in Table 8-15.	No write-back

Table 8-15 lists the data size options supported by all ARCv3 Load and Store instructions, together with the scaling_shift values used by the .AS addressing mode in each case. When double-register Load or Store instructions are used, the scaling shift is the same as for the equivalent single-register Load or Store instruction.

Table 8-15 Scaling Shift

ZZ Suffix	Object Size	scaling_shift	scaling factor
B	byte	0	1
H	half-word	1	2
W	word (32-bit)	2	4
L	long (64-bit)	3	8

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.

8.1.12 Semantics of Load / Store Cache Bypass Mode

Table 8-16 lists the options used for cache bypass modes in Load and Store instructions that provide direct access to main memory instead of accessing the cache hierarchy.

Table 8-16 Cache Bypass Modes

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

8.1.13 Supported Load / Store Data Sizes

Table 8-17 lists the data size modes supported by Load and Store instructions.

Table 8-17 Load Store Data Sizes

Data size mnemonic	Examples	Data size
(none)	LD, ST	Word (32-bit)
B	LDB, STB	Byte (8-bit)
H	LDH, STH	Half-word (16-bit)
L	LDL, STL	Long (64-bit)
DL	LDDL, STDL	Double-word (2x64-bit) Raises an Illegal Instruction exception if M128-OPTION is disabled.

8.1.14 Semantics of Load Data Extension Options

Table 8-18 lists the options for specifying the sign or zero extension semantics of Load instructions. The extension options apply only for instructions whose data size is less than the size of a core register.

Table 8-18 Load Data Extend Mode

X Suffix	Extension Semantics
LD	Data is zero extended.
LD.X	Data is sign extended

8.1.15 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.

8.1.16 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

8.1.16.1 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.

8.1.16.2 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.

8.2 Instruction Syntax Conventions

The mnemonics of instructions encoded in 16-bit formats normally carry a "_S" suffix to distinguish them from 32-bit formats, and where necessary a 32-bit format may carry a "_L" suffix, as shown below:

op	implies 32-bit encoding
op_S	indicates 16-bit encoding

If no suffix is used on the instruction, the implied format is a 32-bit encoded format. Instructions encoded in 16-bit formats have a reduced range of source and target core registers unless indicated otherwise. The notational conventions presented in Table 8-19 are used to describe the syntax of instructions.

Table 8-19 Instruction Syntax Convention

Convention	Description
a	Destination register (reduced range for 16-bit instruction.)
b	Source operand 1 (reduced range for 16-bit instruction) or destination register.
c	Source operand 2 (reduced range for 16-bit instruction) or destination register.
h, g	Full register range for 16-bit instructions; General Purpose register, capable of addressing registers r0-r28 and r31 in some of the 16-bit encoded instructions. When these operands encoded value is 30, the instruction indicates long immediate data rather than r30. And similarly, operand encoded value of 29 is reserved.
A	A 64-bit destination register.
B	A 64-bit source operand register.
C	A 64-bit source operand register.
ACC	Accumulator.
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 setting of ZNCV flags for arithmetic and logical operations
<.aa>	Optional address writeback for load/store operations
<.d>	Optional delay slot mode for branch/jump operations
<.di>	Optional direct data cache bypass for load/store operations
<.x>	Optional sign extension of sub-register load datad
<zz>	Optional load/store data size
u	Unsigned immediate, number indicates field size
s or t	Signed immediate, number indicates field size

Table 8-19 Instruction Syntax Convention (Continued)

limm	Long immediate

Instruction mnemonics with an L suffix perform operations at 64-bit precision and are supported only by the ARC64 ISA. All other instructions perform their operations at 32-bit precision and are supported in all ARCv3 processors.

8.3 Status flags and conditions

The ARCv3 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 ARCv3-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.

8.3.1 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.

8.3.2 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

8.4 Arithmetic and Logical Instructions

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

a ← b op c

a← op b

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

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

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

The normal rules concerning the use of r62 to indicate long-immediate source data and/or a null result apply to these instructions (see "Long Immediate Source Operands and Null Destination Operands" on page 232).

8.4.1 Integer Adder Operations

Table 8-20 summarizes the instructions that perform integer addition, subtraction or comparison operations.

Table 8-20 Integer addition, subtraction and comparison operations

Instruction	Operation	Description
ADD	a ← b + c	add
ADDL	a ← b + c	add
ADC	a ← b + c + C	add with carry
ADCL	a ← b + c + C	add with carry
SUB	a ← b – c	subtract
SUBL	a ← b – c	subtract
SBC	a ← (b – c) - C	subtract with carry
SBCL	a ← (b – c) - C	subtract with carry
CMP	b - c	compare

Table 8-20 Integer addition, subtraction and comparison operations (Continued)

Instruction	Operation	Description
CMPL	b - c	compare
RCMP	c - b	reverse compare
RCMPL	c - b	reverse compare
RSUB	a ← c - b	reverse subtract
RSUBL	a ← c - b	reverse subtract
ADD1	a ← b + (c << 1)	add with left shift by 1
ADD1L	a ← b + (c << 1)	add with left shift by 1
ADD2	a ← b + (c << 2)	add with left shift by 2
ADD2L	a ← b + (c << 2)	add with left shift by 2
ADD3	a ← b + (c << 3)	add with left shift by 3
ADD3L	a ← b + (c << 3)	add with left shift by 3
SUB1	a ← b - (c << 1)	subtract with left shift by 1
SUB1L	a ← b - (c << 1)	subtract with left shift by 1
SUB2	a ← b - (c << 2)	subtract with left shift by 2
SUB2L	a ← b - (c << 2)	subtract with left shift by 2
SUB3	a ← b - (c << 3)	subtract with left shift by 3
SUB3L	a ← b - (c << 3)	subtract with left shift by 3

8.4.1.1 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.

8.4.1.2 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.

8.4.1.3 Comparison Instructions

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

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

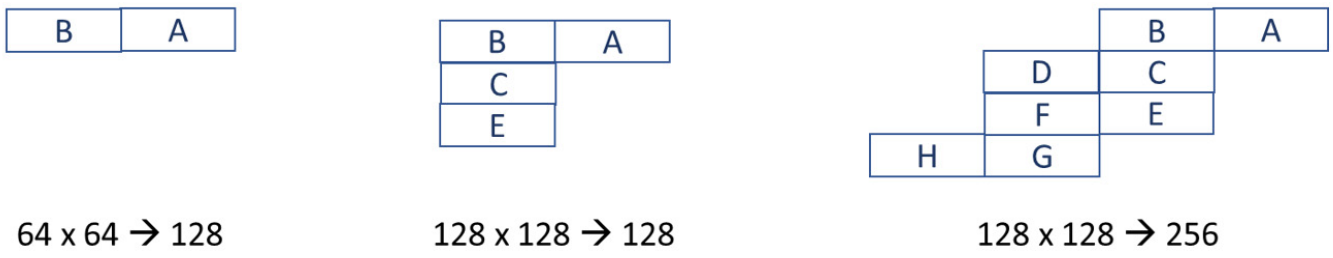
8.4.2 64-bit Integer Multiplication Operations

A minimal set of four 64-bit integer multiply instructions is supported by the ARC64 integer ISA. The MPYL instruction performs 64x64 signed arithmetic returning the lower 64 bits of the product. The MPYML instruction returns the most-significant 64 bits of the signed 128-bit product of two 64-bit values. The MPYMUL and MPYMSUL instructions operate similar to MPYML, except that their operands are {unsigned, unsigned} and {signed, unsigned} respectively.

Together, these multiply instructions can deliver: 64x64→64, signed or unsigned, in one instruction; 64x64→128, signed or unsigned, in two instructions; 128x128→128 in six instructions; and 128x128→256, signed or unsigned, in 14 instructions. The ARC64 ISA needs to provide the capability to compute these large products, but for the ordinary integer ISA there is no requirement to expend large gate counts to implement these operations any faster.

Example code sequences for multi-length multiplication operations are illustrated below in Table 8-21, Table 8-22, Table 8-23 and Table 8-24. For each operation it is assumed that the input operands are X and Y. Where X and Y are 64-bit operands they are presented in registers r0 and r1 respectively. Where X and Y are 128-bit operands they are presented in register pairs {r1, r0} and {r3, r2} respectively. The comments in the right-hand columns indicate the partial calculations A through H illustrated in the Figure 8-2.

Figure 8-2 Components of Multi-length Multiplication Operations



When pairs of multiply operations are used to produce the lower and upper 64-bit halves of a 128-bit partial product, the most-significant half is computed first. This is followed immediately by the lower 64-bit computation to allow the implementation to fuse the two operations together (if supported in HW).

Table 8-21 Signed 64x64 MPY producing a 128-bit Result

```
; 64x64 signed multiplication, producing 128-bit result
   ; o {r3,r2}← r0 x r1
;
mpy_64x64y128_signed:
MPYML r3, r0, r1 // B ← AL x BL (high)
MPYL r2, r0, r1 // A ← AL x BL (low)
```

Table 8-22 Unsigned 64x64 MPY producing a 128-bit Result

```
; 64x64 unsigned multiplication, producing 128-bit result
; o {r3,r2} ← r0 x r1
;
mpy_64x64y128_unsigned:
MPYMUL r3, r0, r1 // B ← AL x BL (high)
MPYL r2, r0, r1 // A ← AL x BL (low)
```

Table 8-23 128x128 MPY producing a 128-bit Result

```
; 128x128 signed multiplication, producing 128-bit result
;  o {r5,r4} ← {r1,r0} x {r3,r2}
;  o requires 1 or 2 temp registers, depending on code schedule
;  o works for both signed and unsigned operands
;
mpy_128x128y128:
      MPYMUL   r5, r0, r2 // B ← AL x BL (high)
      MPYL     r4, r0, r2 // A ← AL x BL (low)
      MPYL     T1, r1, r2 // C ← AH x BL (low)
      ADDL     r5, r5, T1 // B ← B+C
      MPYL     T2, r0, r3 // E ← AL x BH (low)
      ADDL     r5, r5, T2 // B ← B+E
```

Table 8-24 Signed 128x128 MPY producing a 256-bit Result

```
; 128x128 signed multiplication, producing 256-bit result
;  o {r7,r6,r5,r4} ← {r1,r0} x {r3,r2}
;  o requires between 1 and 4 temp registers, depending on code schedule
;
mpy_128x128y256_signed:
      MPYMUL   r5, r0, r2 // B ← AL x BL (high)
      MPYL     r4, r0, r2 // A ← AL x BL (low)
      MPYMSUL  r6, r1, r2 // D ← AH x BL (high)
      MPYL     T1, r1, r2 // C ← AH x BL (low)
      MPYMSUL  T2, r3, r0 // F ← BH x AL (high)
      MPYL     T3, r3, r0 // E ← BH x AL (low)
      MPYML    r7, r1, r3 // H ← AH x BH (high)
      MPYL     T4, r1, r3 // G ← AH x BH (low)
      ADDL.F   r5, r5, T1 // {carry,B} ← B+C
      ADCL.F   r6, r6, T2 // {carry,D} ← D+F+carry
      ADCL     r7, r7, 0  // H ← H + carry
      ADDL.F   r5, r5, T3 // {carry,B} ← B+E
      ADCL.F   r6, r6, T4 // {carry,G} ← D+G+carry
      ADCL     r7, r7, 0  // H ← H + carry
```

8.4.3 Move Operations

Table 8-20 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 ARCv3 processors.

Table 8-25 Move operations

Instruction	Operation	Description
MOV	a ← b	Move source to destination
MOVL	a ← b	Move source to destination
MOVHL	a ← b	Move source to 32-bit high part of long destination
EXTB	a ← extb (b)	Zero-extend byte to word
EXTH	a ← exth (b)	Zero-extend half to word
SEXB	a ← sexb (b)	Sign-extend byte to word
SEXBL	a ← sexbl (b)	Sign-extend byte to long
SEXHL	a ← sexhl (b)	Sign-extend half to long
SEXWL	a ← sexwl(b)	Sign-extend word to long

8.4.3.1 Move Instructions

The primary function of the MOV and MOVL 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 and MOVL_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 11, "Instruction Set Details".

8.4.3.2 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 11, "Instruction Set Details".

8.4.3.3 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.

In ARC64 the LDB.X and LDH.X instructions shall sign-extend their result from the most significant bit of the byte or half-word of load data all the way to bit 63 of the destination register.

Full details of all available formats and encodings are presented in the detailed description of SEXB and SEXHL in Chapter 11, "Instruction Set Details".

8.4.4 Bit-wise Logical Operations

Table 8-26 summarizes the group of instructions that perform bit-wise logical operations. These instructions are always supported in all ARCv3 processors.

Table 8-26 Bit-wise logical operations

Instruction	Operation	Description
AND	a ⟵ b and c	logical bit-wise AND
ANDL	a ⟵ b and c	logical bit-wise AND
OR	a ⟵ b or c	logical bit-wise OR
ORL	a ⟵ b or c	logical bit-wise OR

Table 8-26 Bit-wise logical operations (Continued)

Instruction	Operation	Description
BIC	a ⟵ b and not c	logical bit-wise AND with invert
BICL	a ⟵ b and not c	logical bit-wise AND with invert
XOR	a ⟵ b exclusive-or c	logical bit-wise exclusive-OR
XORL	a ⟵ b exclusive-or c	logical bit-wise exclusive-OR
TST	b and c	bit-wise test, updating flags only
TSTL	b and c	bit-wise test, updating flags only
NOT	a ⟵ ~ b	bit-wise negation
NOTL	a ⟵ ~ b	bit-wise negation

8.4.5 Bit Operations and Mask Operations

8.4.6 Shift and Rotate Operations

Table 8-27 Bit-mask operations

Instruction	Operation	Description
BSET	a ⟵ b or (1<<c)	bit set
BSETL	a ⟵ b or (1<<c)	bit set
BCLR	a ⟵ b and not (1<<c)	bit clear
BCLRL	a ⟵ b and not (1<<c)	bit clear
BTST	b and (1<<c)	bit test
BTSTL	b and (1<<c)	bit test
BXOR	a ⟵ b xor (1<<c)	bit xor
BXORL	a ⟵ b xor (1<<c)	bit xor
BMSK	a ⟵ b and ((1<<(c+1))-1)	bit mask
BMSKL	a ⟵ b and ((1<<(c+1))-1)	bit mask
BMSKN	a ⟵ b and ~((1<<(c+1))-1)	bit mask inverted
BMSKNL	a ⟵ b and ~((1<<(c+1))-1)	bit mask inverted

Table 8-28 Single Bit Shift and Rotate Operations

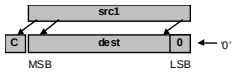
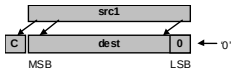
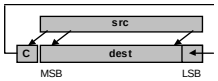
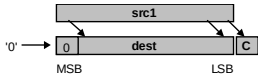
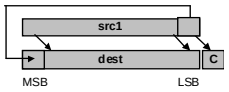
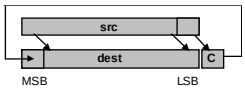
Instruction	Operation	Description
ASL		Arithmetic shift left by one
ASLL		Arithmetic shift left by one
RLC		Rotate left through carry

Table 8-28 Single Bit Shift and Rotate Operations

Instruction	Operation	Description
ASR		Arithmetic shift right by one
ASRL		Arithmetic shift right by one
LSR		Logical shift right by one
LSRL		Logical shift right by one
ROR		Rotate right
RRC		Rotate right through carry

8.4.6.1 Multi-bit Shift Operations

The multi-bit shifter provides a number of instructions that allow any operand to be shifted or rotated, left or right, by up to 32 positions, the result being available for write-back to any core register. Table 8-28 on page 251 lists the single bit shift instructions provided as single operand instructions.

Table 8-29 Multi-bit Shift Operations

Instruction	Operation	Description
ASR multiple		Multiple arithmetic shift right, sign filled

Table 8-29 Multi-bit Shift Operations (Continued)

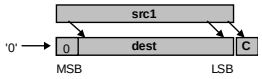
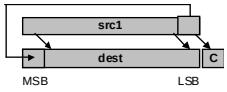
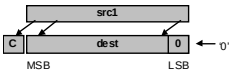
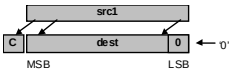
Instruction	Operation	Description
ASRL Multiple		Multiple arithmetic shift right, sign filled
LSR multiple		Multiple logical shift right, zero filled
LSRL Multiple		Multiple logical shift right, zero filled
ROR multiple		Multiple rotate right
ASL Multiple		Multiple arithmetic shift left, zero filled
ASLL Multiple		Multiple arithmetic shift left, zero filled
XBFU	a ⟵ (b >> c[4:0]) and ((1<<(c[9:5]+1))-1)	Extract unsigned bit-field
XBFUL	a ⟵ (b >> c[4:0]) and ((1<<(c[9:5]+1))-1)	Extract unsigned bit-field

Table 8-30 Shift-assist Operations

Instruction	Operation	Description
LSL16	a ⟵ b lsl 16	Logical shift left 16 places
LSR16	a ⟵ b lsr 16	Logical shift right 16 places
LSRL	a ⟵ b lsrl	Logical Shift Right Long
ASLL	a ⟵ b asll	Arithmetic Shift Left Long

Table 8-30 Shift-assist Operations (Continued)

Instruction	Operation	Description
ASRL	a ⟵ b asrl	Arithmetic Shift Right Long
ASR16	a ⟵ b asr 16	Arithmetic shift right 16 places
ASR8	a ⟵ b asr 8	Arithmetic shift right 8 places
LSR8	a ⟵ b lsr 8	Logical shift right 8 places
LSL8	a ⟵ b lsl 8	Logical shift left 8 places
ROL8	a ⟵ b rol 8	Rotate left 8 places
ROR8	a ⟵ b ror 8	Rotate right 8 places

8.4.7 Selection Operations

Table 8-31 Selection operations

Instruction	Operation	Description
MIN	a ⟵ min (b, c)	Select smallest signed integer operand
MINL	a ⟵ min (b, c)	Select smallest signed integer operand (64-bit version)
MAX	a ⟵ max (b, c)	Select largest signed integer operand
MAXL	a ⟵ max (b, c)	Select largest signed integer operand (64-bit version)
ABS	a ⟵ abs (b)	Select absolute value of b
ABSL	a ⟵ abs (b)	Select absolute value of b (64-bit version)

8.4.8 Byte-swapping Operations

Table 8-32 Byte-swapping operations

Instruction	Operation	Description
SWAP	b ⟵ swap (c)	Swap words
SWAPL	b ⟵ swap (c)	Swap words (64-bit version)
SWAPE	b ⟵ swap_endian (c)	Swap endianness
SWAPEL	b ⟵ swap (c)	Swap words(64-bit version)

8.4.9 Bit-scanning Operations

Table 8-33 Bit-scanning operations

Instruction	Operation	Description
NORM	b ⟵ norm (c)	Normalize
NORML	b ⟵ norm (c)	Normalize (64-bit version)
NORMH NORMW	b ⟵ norm (c)	Normalize word
FFS	a ⟵ b ffs (c)	Find first set
FFSL	a ⟵ b ffs (c)	Find first set
FLS	a ⟵ b fls (c)	Find last set
FLSL	a ⟵ b fls (c)	Find last set

8.4.10 Relational Comparison Operations

Table 8-34 Relational Comparison Operations

SETcc variant	Operation	Description
SETEQ	a ⟵ (b == c)	Set if equal
SETNE	a ⟵ (b != c)	Set if not equal
SETLT	a ⟵ (b < c) signed	Set if less than
SETGE	a ⟵ (b >= c) signed	Set if greater than or equal
SETLO	a ⟵ (b < c) unsigned	Set if lower than
SETHS	a ⟵ (b >= c) unsigned	Set if higher or same
SETLE	a ⟵ (b ≤ c) signed	Set if less than or equal
SETGT	a ⟵ (b > c) signed	Set if greater than

Table 8-35 Relational Comparison for 64-bit Operations

SETccL variant	Operation	Description
SETEQL	a ⟵ (b == c)	Set if equal
SETNEL	a ⟵ (b != c)	Set if not equal
SETLTL	a ⟵ (b < c) signed	Set if less than
SETGEL	a ⟵ (b >= c) signed	Set if greater than or equal
SETLOL	a ⟵ (b < c) unsigned	Set if lower than

Table 8-35 Relational Comparison for 64-bit Operations (Continued)

SETccL variant	Operation	Description
SETHSL	a ⟵ (b >= c) unsigned	Set if higher or same
SETLEL	a ⟵ (b ≤ c) signed	Set if less than or equal
SETGTL	a ⟵ (b > c) signed	Set if greater than

8.4.11 Integer Multiply, Multiply-accumulate, and Divide Operations

Table 8-36 Integer Multiply, MAC and Divide Operations

Instruction	Operation	Description
MPY, MPY_S	a ⟵ b * c	32 x 32 signed integer multiply, returning lower 32 bits of product
MPYL	a ⟵ b * c	64 x 64 signed integer multiply, returning lower 64 bits of product
MPYU	a ⟵ b * c	32 x 32 unsigned integer multiply, returning lower 32 bits of product
MPYMUL	a ⟵ b * c	64 x 64, unsigned x unsigned, retuning upper 64 bits of product
MPYMSUL	a ⟵ b * c	64 x 64, signed x unsigned, returning upper 64 bits of product
MPYM MPYH	a ⟵ b * c	32 x 32 signed integer multiply, returning upper 32 bits of product
MPYML	a ⟵ b * c	64 x 64, upper 64 bits of product
MPYMU MPYHU	a ⟵ b * c	32 x 32 unsigned integer multiply, returning upper 32 bits of product
MPYW, MPYW_S	a ⟵ b * c	16 x 16 signed integer multiply, returning 32 bit product
MPYUW, MPYUW_S	a ⟵ b * c	16 x 16 unsigned integer multiply, returning 32 bit product
MPYD	a ⟵ b * c	32 x 32 signed integer multiply, returning 64-bit result
MPYDU	a ⟵ b * c	32 x 32 unsigned integer multiply, returning 64-bit result
MAC	acc ⟵ acc + (b * c) a⟵ acc_lo	32 x 32 signed integer multiply-accumulate, returning lower 32 bits of the accumulated results
MACU	acc ⟵ acc + (b * c) a⟵ acc_lo	32 x 32 unsigned integer multiply-accumulate, returning lower 32 bits of the accumulated results
MACD	acc ⟵ acc + (b * c) a⟵ acc_lo	32 x 32 signed integer multiply-accumulate, returning 64 bits of the accumulated results
MACDU	acc ⟵ acc + (b * c) a⟵ acc_lo	32 x 32 unsigned integer multiply-accumulate, returning 64 bits of the accumulated results
DIV	a ⟵ b / c	Signed integer division, returning a 32-bit quotient
DIVL	a ⟵ b / c	Signed integer division, returning a 32-bit quotient

Table 8-36 Integer Multiply, MAC and Divide Operations

Instruction	Operation	Description
DIVU	a ⟵ b / c	Unsigned integer division, returning a 32-bit quotient
DIVUL	a ⟵ b / c	Unsigned integer division, returning a 64-bit quotient
REM	a ⟵ b % c	Signed integer division, returning the 32-bit remainder
REML	a ⟵ b % c	Signed integer division, returning the 64-bit remainder
REMU	a ⟵ b % c	Unsigned integer division, returning the 32-bit remainder
REMUL	a ⟵ b % c	Unsigned integer division, returning the 64-bit remainder

8.4.12 Dual and Quad Integer Multiply / Accumulate Operations

Table 8-37 Dual and Quad Integer Multiply and MAC Operations

Instruction	Operation	Description
DMPYH	`if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}`	Dual 16x16 multiply
DMPYHU	if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}	Dual 16x16 multiply unsigned

DMACH	if (cc) {
	result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
	a = result.w0;
	acc = result;
	}
		Dual 16x16 MAC

DMACHU	if (cc) {
	result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
	a = result.w0;
	acc = result;
	}
		Dual 16x16 MAC unsigned

QMPYH	if (cc) {
	result = (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3);
	A = result;
	acc = result;
	}
		Quad 16x16 multiply

Table 8-37 Dual and Quad Integer Multiply and MAC Operations

Instruction	Operation	Description

QMPYHU	if (cc) {
	result = (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3);
	A = result;
	acc = result;
	}
		Unsigned QMPYH

DMPYWH	if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}	Dual 32x16 multiply
DMPYWHU	if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}	Unsigned dual 32x16 multiply

QMACH	if (cc) {
	result = acc + (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3);
	A = result;
	acc = result;
	}
		Quad 16x6 MAC

QMACHU	if (cc) {
	result = acc + (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3);
	A = result;
	acc = result;
	}
		Unsigned quad 16x16 MAC

DMACWH	if (cc) {
	result = acc +((B.w0 * c.h0)+(B.w1 * c.h1));
	A = result;
	acc = result;
	}
		Dual 32x16 MAC

DMACWHU	if (cc) {
	result = acc +((B.w0 * c.h0)+(B.w1 * c.h1));
	A = result;
	acc = result;}
		Unsigned 32x16 MAC

8.4.13 Integer Vector Operations

Table 8-38 Integer Vector ADD, SUB, MPY operations

Instruction	Operation	Description

VADD2H	if (cc) {
	a.h0 = b.h0 + c.h0;
	a.h1 = b.h1 + c.h1;
	}
		Vector-add, dual 16-bit

VSUB2H	if (cc) {
	a.h0 = b.h0 - c.h0;
	a.h1 = b.h1 - c.h1;
	}
		Vector-sub, dual 16-bit

VADDSUB2H	if (cc) {
	a.h0 = b.h0 + c.h0;
	a.h1 = b.h1 - c.h1;
	}
		Vector-add/sub, dual 16-bit

VSUBADD2H	if (cc) {
	a.h0 = b.h0 - c.h0;
	a.h1 = b.h1 + c.h1;
	}
		Vector-sub/add, dual 16-bit

VMPY2H	if (cc) {
	acc.w0 = A.w0 = b.h0 * c.h0;
	acc.w1 = A.w1 = b.h1 * c.h1;
	}
		Vector-multiply, dual 16x16

VMPY2HU	if (cc) {
	acc.w0 = A.w0 = b.h0 * c.h0;
	acc.w1 = A.w1 = b.h1 * c.h1;
	}
		Unsigned vector-multiply, dual 16x16

VADD4H	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;
	}
		Vector-add, quad 16-bit

VSUB4H	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;
	}
		Vector-sub, quad 16-bit

Table 8-38 Integer Vector ADD, SUB, MPY operations

Instruction	Operation	Description

VADDSUB4H	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;
	}
		Vector add/sub, quad 16-bit

VSUBADD4H	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;
	}
		Vector-sub/add, quad 16-bit

VADD2	if (cc) { A.w0 = B.w0 + C.w0; A.w1 = B.w1 + C.w1;}	Vector-add, dual 32-bit

VSUB2	if (cc) {
	A.w0 = B.w0 - C.w0;
	A.w1 = B.w1 - C.w1;
	}
		Vector-sub, dual 32-bit

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

8.5 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 8-39 Memory Access Instructions

Instruction	Operation	Description
EX	b ⟵ mem[c]; mem[c] b	Atomic Exchange

Table 8-39 Memory Access Instructions (Continued)

Instruction	Operation	Description
EXL	b ⟵ mem[c]; mem[c] b	64-bit Atomic Exchange
LD(LD LDH LDW LDB LDL LDDL)	Load register-register	see "General Operations Register-register Format" on page 289
LLOCK	b ⟵ mem[c]; LF ⟵ 1;	Load locked
LLOCKL	B ⟵ mem[C]; LF ⟵ 1;	Load locked on 64-bit data
SCONDL	if LF mem[C] ♦ B	Store conditional on 64-bit data

The transfer of data to and from memory is accomplished with the load and store commands (LD, ST). These instructions can write the result of the address computation back to the address source register, pre or post calculation. This operation is accomplished with the optional address write-back suffices: .A or .AW (register updated pre-memory transaction), and .AB (register updated post memory transaction). Addresses are interpreted as byte addresses unless the scaled address mode is used, as indicated by the address suffix .AS. The scaled address mode does not write back the result of the address calculation to the address source register.

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).

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.

8.5.1 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/LDL/LDDL<zz><.x><.aa><.di>	a,[b]	*(uses ld a,[b,0])*
LD/LDL/LDDL<zz><.x><.aa><.di>	a,[b,s9]	
LD/LDL/LDDL<zz><.x><.di>	a,[limm,s9]	*(Redundant format, use ld a,[limm])*
LD/LDL/LDDL<zz><.x><.di>	a,[limm]	*(= ld a,[limm,0])*

LD/LDL/LDDL<zz><.x><.aa><.di>	a,[b,c]	
LD/LDL/LDDL<zz><.x><.aa><.di>	a,[b,limm]	
LD/LDL/LDDL<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.AS	a, [b, c]	*(reduced set of regs)*
LD_S	r1, [GP,s9]	*(reduced set of regs)*

8.5.2 Store Instructions

8.5.2.1 Store Register with Offset

The following is the syntax for the Store register with offset instruction:

ST/STL/STDL<zz><.aa><.di>	w6,[b]	*(use st w6,[b,0])*
ST/STL/STDL<zz><.aa><.di>	w6,[b,s9]	
ST/STL/STDL<zz><.di>	w6,[limm]	*(= st w6,[limm,0])*
ST/STL/STDL<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	w6, [b, u7]	*(u7 offset is 32-bit aligned)*

STB_S	w6, [b, u5]
STH_S	w6, [b, u6] *(u6 offset is 16-bit aligned)*

8.5.3 Stack Pointer Operations

ARCv3-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)*
PUSHL_S	b	*(Push b onto the stack)*
PUSHDL_S	b	*(Push b onto the stack)*

The following is the instructions syntax for pop (POPL_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)*
POPL_S	b	*(Pop top item of stack to b)*
POPDL_S	b	*(Pop top item of stack to b)*

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.

8.5.4 Atomic Memory Operations

The data size of the baseline ARCv3 AMOs is 32-bits. The ARC64 ISA also includes an equivalent set of 64-bit AMOs, which operate on 64-bit memory values.

The standard C/C++ atomic library defines a number of atomic *fetch-and-operate* functions that are most efficiently implemented when there is support for those operations in the instruction set. The ARCv3 ISA therefore provides instructions to support these atomic library operations. Support for atomic swap already exists in the ARCv3 ISA, in the form of the EX instruction. Eight further atomic memory operations (AMOs) are included in ARCv3; ADD, OR, AND, XOR, and both signed and unsigned MIN and MAX operations.

Note MINU and MAXU are the unsigned equivalents of the signed MIN and MAX operations. All operations are performed on 32-bit values, although the ARC64 ISA introduces an additional set of equivalent AMO instructions that operate on 64-bit values. The read-modify-write operation implied by these instructions is performed atomically, that is, there is never any other read or write to the memory location addressed by these instructions occurring between the read or write operations for these instructions.

Each of the AMOs provides an option to specify acquire semantics on the memory read and release semantics on the memory write. This is indicated by the optional suffix `.aq.rl` applied to the instruction mnemonic.

The ARCv3 ISA continues to support the uncached version of the EX instruction (EX.DI), but within ARCv3 the ARC64 ISA does not provide uncached versions of the new AMOs nor an uncached version of EXL.

All AMOs perform an atomic sequence in which the memory location defined by Rc is read; the value read is then combined with source register Rb using the operator of that instruction; the result is written back to memory at address Rc; and the original value from memory is returned to destination register Rb. The EX instruction is a degenerate form of AMO in which the value written to memory is simply the Rb source operand. This has the effect of swapping the contents of Rb and the contents of memory at the location given by Rc.

The memory address of an AMO must be aligned to the size of the data object it references. This is primarily to ensure that the object referenced by an atomic memory operation never straddles a cache-line boundary. If an AMO has a non-aligned address it always raises an EV_Misaligned exception.

Table 8-40 Atomic Memory Operations (AMOs) - 32 bit

Instruction		Semantics
`ex<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ←mem[Rc]; mem[Rc] ← Rb; Rb ← tmp)
`atld.add<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ← mem[Rc]; mem[Rc] = add(Rb, mem[Rc]); Rb ← tmp)
`atld.or<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ← mem[Rc]; mem[Rc] = or(Rb, mem[Rc]); Rb ← tmp)
`atld.and<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ← mem[Rc]; mem[Rc] = and(Rb, mem[Rc]); Rb ← tmp)
`atld.xor<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ← mem[Rc]; mem[Rc] = xor(Rb, mem[Rc]); Rb ← tmp)
`atld.minu<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ← mem[Rc]; mem[Rc] = umin(Rb, mem[Rc]) ; Rb ← tmp)
`atld.maxu<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ← mem[Rc]; mem[Rc] = umax(Rb, mem[Rc]) ; Rb ← tmp)
`atld.min<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ← mem[Rc]; mem[Rc] = min(Rb, mem[Rc]) ; Rb ← tmp)
`atld.max<.aq.rl>`	`Rb,[Rc]`	atomic (tmp ← mem[Rc]; mem[Rc] = max(Rb, mem[Rc]) ; Rb ← tmp)

8.5.4.1 Instruction Encodings for 32-bit AMOs

The 32-bit AMOs are encoded in the same format as the existing EX instruction, that is using the REG_REG form of the SOP sub-format of the F32_GEN4 major format as shown in Table 8-41.

Table 8-41 Encodings for 32-bit AMO Instructions

`ex`	`b,[c]`	`00100bbb00101111 0BBBCCCCCC001100`
`ex`	`b,[u6]`	`00100bbb01101111 0BBBCCCCCC001100`

```
ex.di                   b,[c]     00100bbb00101111 1BBBCCCCCC001100
ex.di                   b,[u6]    00100bbb01101111 1BBBCCCCCC001100
ex.aq.rl                b,[c]     00100bbb00101111 0BBBCCCCCC001110
ex.aq.rl                b,[u6]    00100bbb01101111 0BBBCCCCCC001110
atld<.op><.aq.rl>       b,[c]     00100bbb00101111 mBBBCCCCCC110iii
```

Table 8-42 ATLD Sub-Opcode Encodings

I[2:0]	Atomic Load Instruction
000	atld.add
001	atld.or
010	atld.and
011	atld.xor
100	atld.minu
101	atld.maxu
110	atld.min
111	atld.max

8.5.4.2 64-bit AMO

The baseline ARCv3 ISA includes 64-bit atomic memory operations that supports the standard set of C11/C++11 atomic operations, when `-atomic_option = 2`.

Table 8-43 Atomic Memory Operations (AMOs) - 64 bit

Instruction		Semantics
exl<.aq.rl>	Rb,[Rc]	atomic (tmp ←mem[Rc]; mem[Rc] ← Rb; Rb ← tmp)
atldl.add<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] += (Rb, mem[Rc]); Rb ← tmp)
atldl.or<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] |= (Rb, mem[Rc]); Rb ← tmp)
atldl.and<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] &= (Rb, mem[Rc]); Rb ← tmp)
atldl.xor<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] ^= (Rb, mem[Rc]); Rb ← tmp)
atldl.minu<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = umin(Rb, mem[Rc]) ; Rb ← tmp)
atldl.maxu<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = umax(Rb, mem[Rc]) ; Rb ← tmp)
atldl.min<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = min(Rb, mem[Rc]) ; Rb ← tmp)
atldl.max<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = max(Rb, mem[Rc]) ; Rb ← tmp)

The 64-bit AMOs supported by the ARC64 ISA, are encoded in the same format as the EXL instruction using the REG_REG form of the SOP sub-format of the F32_GEN_OP64 major format as shown in Table 8-44.

Table 8-44 Encodings for 64-bit AMO Instructions

exl	b,[c]	01011bbb00101111 0BBBCCCCCC001100
exl.aq.rl	b,[c]	01011bbb00101111 0BBBCCCCCC001110
atldl<.op><.aq.rl>	b,[c]	00100bbb00101111 mBBBCCCCCC110iii

The 3-bit sub-opcode field <.op> is encoded in the I[2:0] field from bits [2:0] of the instruction, as shown in Table 8-42 and Table 8-45.

Table 8-45 ATLD Sub-Opcode Encodings

I[2:0]	Atomic Load Instruction
000	atldl.add
001	atldl.or
010	atldl.and
011	atldl;.xor
100	atldl.minu
101	atldl.maxu
110	atldl.min
111	atldl.max

The m field (bit 15) encodes the <.aq.rl> option for the new AMOs (add, or, and, xor, min, max, umin, umax). The <.aq.rl> option is enabled when m=1 and disabled when m=0.

Any value of the operand format bits [23:22] other than 00 is illegal for the new AMOs, and raises an Illegal Instruction exception.

In common with the legacy EX instructions, an Illegal Instruction is raised if a LIMM operand is specified as the b register operand of all new EX and ATLD. Both the signed and unsigned LIMM operands are similarly Illegal as b register operands in the 64-bit versions of all EXL and ATLDL instructions in ARC64.

The legacy ARCv3 EX instructions support the [u6] operand format, although it is of very limited value. These formats continue to be supported in ARCv3 for compatibility reasons, but the [u6] format is not supported for the new ARC64 AMOs (including EXL, LLOCKL and SCONDL).

8.5.5 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. Refer to the relevant core-specific Databook for details:

- PREFETCH: Prefetch data with no intention to write.

- PREFETCHW: Prefetch data with intention to write, that is, the line is allocated in dirty/modified state in the data cache. This option cannot be configured in the ARC EM processor, and hence this instruction is decoded as an illegal instruction in the ARC EM processor.
- 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 hence 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 ARC32-based processors andr62 or r60 in the ARC64-based processors.

Table 8-46 PREFETCH, PREFETCHW, AND PREALLOC Encodings

{X ZZ} Fields	{ AA DI } Fields							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 01								
1 01								
0 10								
1 10								
0 00	PREFETCH	PREFETCHW.DI	PREFETCH.AW	PREFETCHW.AW	PREFETCH.AB	PREFETCHW.AB	PREFETCH.AS	PREFETCHW.AS
1 00	PREALLOC			PREALLOC.AW		PREALLOC.AB		PREALLOC.AS
0 11								
1 11								

8.6 Auxiliary Register Operations

Table 8-47 Auxiliary Register Operations

Instruction	Operation	Description
AEX	tmp ⟵ aux.reg[c] aux.reg[c] ⟵ b b ⟵ tmp	Auxiliary register exchange with core register
AEXL	tmp ⟵ aux.reg[c] aux.reg[c] ⟵ b b ⟵ tmp	Auxiliary register exchange with core register
LR	b ⟵ aux.reg[c]	Load from auxiliary register
LRL	b ⟵ aux.reg[c]	Load long from auxiliary register
SR	aux.reg[c] ⟵ b	store to auxiliary register
SRL	aux.reg[c] ⟵ b	store long to auxiliary register

8.6.1 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" on page 289, the "General Operations Register with Unsigned 6-bit Immediate" on page 289, and the "General Operations Register with Signed 12-bit Immediate" on page 289.

8.6.2 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" on page 289, the "General Operations Register with Unsigned 6-bit Immediate" on page 289, and the "General Operations Register with Signed 12-bit Immediate" on page 289.

8.6.3 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" on page 289, the "General Operations Register with Unsigned 6-bit Immediate" on page 289, the "General Operations Register with Signed 12-bit Immediate" on page 289, and the "General Operations Conditional Register with Unsigned 6-bit Immediate" on page 290.

8.7 Control Flow Instructions

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

Table 8-48 Control Flow Operations

Instruction	Operation	Description
J	pc ⟵ c	Unconditional jump
Jcc	if (cc) pc ⟵ c	Conditional jump
JL	blink ⟵ next_pc; pc ⟵ c	Jump and link

JLcc	if (cc) {
	  blink ⟵ next_pc;
	  pc ⟵ c
	}
		Jump and link conditionally

JLI_S	blink ⟵ next_pc; pc ⟵ JLI_BASE + (u10 << 2);	Jump and link indexed
B B_S	pc ⟵ pcl+ offset	Branch unconditionally

Bcc
Bcc_S
	if (cc) {
	  pc ⟵ pcl + offset
	}
		Branch conditionally

Table 8-48 Control Flow Operations

Instruction	Operation	Description

BLcc	if (cc) {
	blink ⟵ next_pc;
	pc ⟵ pcl + offset
	}
		Branch-and-link conditionally

BRcc	if (cmp(b,c)) {
	pc ⟵ pcl + offset
	}
		Compare-and-branch

BRccL	if (cmp(b,c)) {
	pc ⟵ pcl + offset
	}
		Compare-and-branch

BBIT0	if (b[c] == 0) {
	pc ⟵ pcl + offset
	}
		Branch if bit is zero

BBIT0L	if (b[c] == 0) {
	pc ⟵ pcl + offset
	}
		Branch if bit is zero

BBIT1	if (b[c] == 1) {
	pc ⟵ pcl + offset
	}
		Branch if bit is one

BBIT1L	if (b[c] == 1) {
	pc ⟵ pcl + offset
	}
		Branch if bit is one

BI	pc ⟵ next_pc + (c << 2)	Branch indexed
BIH	pc ⟵ next_pc + (c << 1)	Branch indexed half-word

DBNZ	if (src != 1) {
	PC ⟵ PCL + offset
	}
	src = src - 1
		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.
ENTER	Function prolog sequence	This instruction executes the function prolog code, and is typically used by functions that save both integer and floating-point callee-saved registers to the stack.

Table 8-48 Control Flow Operations

Instruction	Operation	Description
LEAVE	Function epilog sequence	This instruction executes the function epilogue code, and is used by functions that wish to restore both integer and floating-point callee-saved registers from the stack.
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.

8.7.1 Branch Instructions

Because of the pipelined nature of the processor, a branch instruction might not take effect immediately, but after a delay of one or more cycles, depending on the implementation. The execution of the immediately following instruction after the branch can be enabled in order to minimize the cost of each branch instruction. Such a following instruction is said to be in the *delay slot*. The modes for specifying the execution of the delay slot instruction are indicated by the optional .d field according to the Table 8-49.

Table 8-49 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 8-9 on page 233 lists the condition codes that are available for conditional branch instructions.

8.7.1.1 Conditional Branch Instructions

Conditional Branch (Bcc) has a branch range of ±1MB, whereas unconditional branch (B) has larger range of ±16MB. 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
```

8.7.1.2 Branch and Link Instructions

Conditional Branch and Link (BLcc) has a branch range of ±1MB, whereas unconditional Branch-and-Link (BL) has larger range of ±16MB. 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".

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)
```

8.7.1.3 Branch On Compare or Bit Test

Branch on Compare (BRcc) and Branch on Bit Test (BBIT0, BBIT1) have a branch range of ±256B. The branch target address is 16-bit aligned.

The BRcc instruction is similar in execution to a normal compare instruction (CMP) with the addition that a branch occurs if the condition is met. No flags are updated and no ALU result is written back to the register file. A limited set of condition code tests are available for the BRcc instruction as shown in the following table. Table 8-50 lists the additional condition code tests that are available through the effect of reversing the operands.

Table 8-50 Branch on Compare/Test Mnemonics

Mnemonic	Condition
BREQ	Branch if b and c are equal
BREQL	Branch if b and c are equal (64-bit instruction)
BRNE	Branch if b and c are not equal

Table 8-50 Branch on Compare/Test Mnemonics (Continued)

Mnemonic	Condition
BRNEL	Branch if b and c are not equal (64-bit instruction)
BRLT	Branch if b is less than c
BRLTL	Branch if b is less than c (64-bit instruction)
BRGE	Branch if b is greater than or equal to c
BRGEL	Branch if b is greater than or equal to c (64-bit instruction)
BRLO	Branch if b is lower than c
BRLOL	Branch if b is lower than c (64-bit instruction)
BRHS	Branch if b is higher than or same as c
BRHSL	Branch if b is higher than or same as c
BBIT0	Branch if bit c in register b is clear
BBIT0L	Branch if bit c in register b is clear (64-bit instruction)
BBIT1	Branch if bit c in register b is set
BBIT1L	Branch if bit c in register b is set (64-bit instruction)

Table 8-51 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)
BRGTL 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)
BRLEL 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)
BRHIL 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)
BRLSL b,c,s9	Branch if b is lower than or same c (encode as BRHS c,b,s9)

Table 8-52 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 8-52 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)
BRGTL 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)
BRLEL 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)
BRHIL b,u6,s9	Branch if b is higher than u6 (encode as BRHS b,u6+1,s9)
BRLS b,u6,s9	Branch if if b is lower than or same u6 (encode as BRLO b,u6+1,s9)
BRLSL b,u6,s9	Branch if if b is lower than or same u6 (encode as BRLO b,u6+1,s9)

8.7.2 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.

8.8 Vector Pack Instructions

This section details the vector pack instructions.

Table 8-53 Special Instructions

Instruction	Operation	Description
VPACK4HL	`a = (c.h2 << 48) | (c.h0 << 32) | (b.h2 << 16) | b.h0`	Compose the destination result using the even-numbered half-words from the source operands
VPACK4HM	`a = (c.h3 << 48) | (c.h1 << 32) | (b.h3 << 16) | b.h1`	Compose the destination result using the odd-numbered half-words from the source operands
VPACK2WL	`a = (c.w0 << 32) | b.w0`	Compose the destination result using the even-numbered words from the source operands
VPACK2WM	`a = (c.w1 << 32) | b.w1`	Compose the destination result using the odd-numbered words from the source operands
VPACK2HL	`a = (b.h0 << 16) | c.h0`	Compose the destination operand from the lower 16-bits of the source operands.

Table 8-53 Special Instructions (Continued)

Instruction	Operation	Description
VPACK2HM	`a = (b.h1 << 16) | c.h1`	Compose the destination operand from the higher 16-bits of the source operands.

8.9 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 8-54 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
WAIT	Wait for a wake-up event of any type. Enter sleep state and wait for a wakeup event	Wait for a wake-up event of any type. Enter sleep state and wait for a wakeup event
WEVT	Enter the sleep state and wait for an event	Enter the sleep state and wait for an event
WLFC	Enter the sleep state to reduce dynamic power during busy-waiting loops	Enter the sleep state to reduce dynamic power during busy-waiting loops
SYNC	Synchronize with memory	Synchronize with memory

Syntax for special instructions:

Some instructions require no source operands or destinations. The ARCv3 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

8.9.1 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.

8.9.2 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 only when all previous instructions have completed, after which it flushes the pipeline.

8.9.3 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.

8.9.4 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
```

8.9.5 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
```

8.9.6 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.

8.9.7 Synchronization Instructions

The synchronize instruction, SYNC, waits until all previous instructions (LD, ST, cache fills) have completed all of their actions, retired their results, and/or have written any store data to memory. The DSYNC instruction waits until all previous memory operations have similarly completed their actions. The DMB instruction is described separately.

8.10 APEX Extension Instructions

These operations are generally of the form **a ← b 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.

8.10.1 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
```

8.10.2 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
```

8.10.3 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

8.11 Serializing Instructions and Events

This section defines and explains the semantics of **serializing instructions** and **serializing events**. These definitions are important in the correct implementation of operations such as halting the core, entering sleep mode, raising an imprecise exception, and executing certain supervisor instructions that require the core to complete all previous operations.

8.11.1 Instruction Barrier

An instruction barrier (IB) is an event that occurs between two successive instructions in the execution sequence on a given core that prevents any overlap of instructions on that core before the IB with instructions after the IB. This can be defined more precisely by observing the state updates from instructions before and after the IB. An IB imposes two restrictions:

- All register and flag updates in the local core, from instructions before the IB, must be visible to all instructions in the local core after the IB, and;
- All memory updates in the local core, from instructions before the IB, must be visible to all other cores before any instruction in the local core after the IB observes any memory locations.

Suppose all instructions before the IB have committed and completed all post-commit updates before the IB. In that case, this design ensures that all updates from instructions before the IB are visible to all instructions after the IB is considered to be complete. These updates include:

- All updates to registers, flags, and memory
- Taking the exception and the resulting updates to exception auxiliary registers and PC when an instruction triggers an exception. Such exceptions might be imprecise and may occur sometime after the offending instruction commits; the IB must wait for all pending write buffer acknowledgments to ensure that any associated imprecise exceptions are taken before the IB completes.

Taking an exception means performing only the exception entry sequence, and it does not include executing any handler instructions.

8.11.2 Serializing Instructions

A serializing instruction (SI) is an instruction preceded by an IB and succeeded by an IB. The SI, therefore, does not commit until after its preceding IB. Following are the implications of serializing semantics:

- there can be no state updates from instructions before the SI occurring after the SI, and
- there can be no state updates from after the SI occurring before the SI, and

- all updates made by the SI itself are visible to all instructions after the SI.

8.11.3 Serializing Events

When a serializing event (SE) occurs in the local core between two successive instructions I1 and I2, an implicit IB is inserted between I1 and I2. An SE is typically asynchronous to the execution sequence. Hence, the actual instructions that bracket the SE depend on the actual time at which the SE is recognized. I1 is the last instruction to commit before the SE is recognized, and I2 is the first instruction to commit after the SE completes all of its state updates.

8.11.4 Serializing Instructions in ARCv2 and ARCv3

The following instructions (and their compact 16-bit equivalent encodings) are serializing instructions: **SYNC, TRAP, RTIE**, **BRK**, **WEVT**, **WLFC**, **WAIT**, **SLEEP**, **CLRI**, **SETI**, **KFLAG**, and **FLAG**.

8.11.5 Serializing Events in ARCv2 and ARCv3

Following are the serializing events:

- external halt or run requests
- an external Actionpoint trigger
- an ARC event input

9

32-bit Instruction Formats Reference

This chapter explains the 32-bit instruction formats of ARCv3 instruction set architecture. The chapter on "Instruction Set Summary" on page 223 lists and summarizes the 32-bit formats. In particular, Table 8-3 illustrates all of the 32-bit top-level formats of the ARCv3 ISA in a concise form, and may be used as a reference. The list of syntax conventions is shown in Table 8-19, and section "Encoding Notation" on page 230 describes the encoding notation used when describing instruction encodings in this chapter.

This chapter describes the following 32-bit ARCv3 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 8, "Instruction Set Summary".

For full details on the available encodings and syntax for each instruction, see the relevant page for each instructions in Chapter 11, "Instruction Set Details".

9.1 Branch Format [F32_BR0]

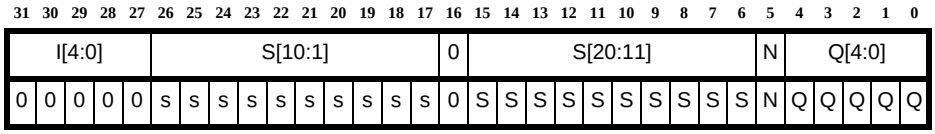
The F32_BR0 format is used to encode branch instructions with large displacements. There are two sub-formats, COND and UCOND_FAR. The COND format is used to encode conditional branch instructions, with an optional delay slot, and the UCOND_FAR format is used to encode unconditional branches with maximum possible displacement and an optional delay slot.

For all instructions in this format the branch target address is 16-bit aligned, as all ARCv3 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.

9.1.1 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 8-10. For information about the encoding of delay-slot modes, see Table 8-13.

Figure 9-1 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.

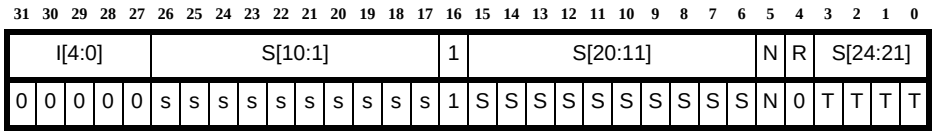
Syntax:

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

9.1.2 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 8-13 for information on delay slot mode encoding.

Figure 9-2 Branch Unconditional Far Format



Syntax:

```
B<.d>  s25   (unconditional branch far)
```

9.2 BRcc, BBITn and BL Format [F32_BR1]

This format encodes a set of branch instructions that include a comparison within the instruction. The BRccand BRccL instructions perform a relational test between two source operands and branch if the relation is true. The BBIT0,BBIT0L, BBIT1,and BBIT1L 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.

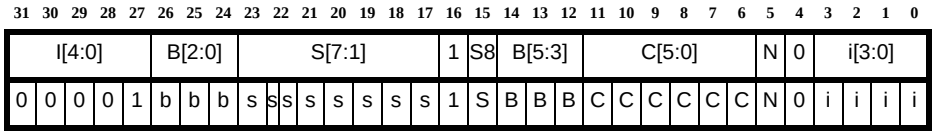
These instructions may have an optional delay-slot. For information about the delay slot mode encoding, see Table 8-13.

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).

9.2.1 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 ARCv3 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 9-3 Branch on Compare Register-register Format



Syntax:

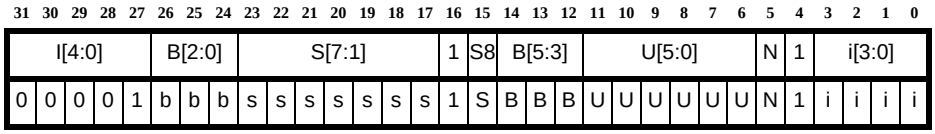
BRcc<.d>	b,c,s9	*(branch if reg-reg compare is true, swap regs if inverse condition required)*
BRcc	b,limm,s9	*(branch if reg-limm compare is true)*
BRcc	limm,c,s9	*(branch if limm-reg compare is true)*
BRccL<.d>	b,c,s9	*(branch if reg-reg compare is true, swap regs if inverse condition required)*
BRccL	b,limm,s9	*(branch if reg-limm compare is true)*
BRccL	limm,c,s9	*(branch if limm-reg compare is true)*
BBIT0<.d>	b,c,s9	*(branch if bit c in reg b is clear)*
BBIT0	b,limm,s9	*(branch if reg-limm compare is true)*
BBIT0	limm,c,s9	*(branch if limm-reg compare is true)*
BBIT0L<.d>	b,c,s9	*(branch if bit c in reg b is clear)*
BBIT0L	b,limm,s9	*(branch if reg-limm compare is true)*
BBIT0L	limm,c,s9	*(branch if limm-reg compare is true)*
BBIT1<.d>	b,c,s9	*(branch if bit c in reg b is set)*
BBIT1	b,limm,s9	*(branch if reg-limm compare is true)*

BBIT1	limm,c,s9	*(branch if limm-reg compare is true)*
BBIT1L<.d>	b,c,s9	*(branch if bit c in reg b is set)*
BBIT1L	b,limm,s9	*(branch if reg-limm compare is true)*
BBIT1L	limm,c,s9	*(branch if limm-reg compare is true)*

9.2.2 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 ARCv3 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 9-4 Branch on Compare Register-Immediate Format



Syntax:

BRcc<.d>	b,u6,s9	*(branch if reg-immediate compare is true, use "immediate+1" if a missing condition is required)*
BRccL<.d>	b,u6,s9	*(branch if reg-immediate compare is true, use "immediate+1" if a missing condition is required)*
BBIT0<.d>	b,u6,s9	*(branch if bit u6 in reg b is clear)*
BBIT0L<.d>	b,u6,s9	*(branch if bit u6 in reg b is clear)*
BBIT1<.d>	b,u6,s9	*(branch if bit u6 in reg b is set)*
BBIT1L<.d>	b,u6,s9	*(branch if bit u6 in reg b is set)*

Note This format also supports operand options where b is replaced by limm.

9.2.3 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.

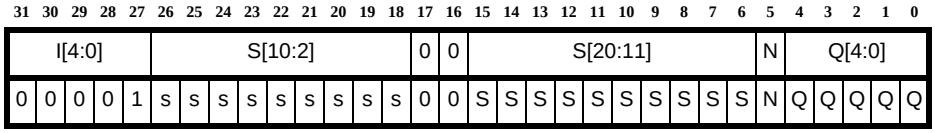
The branch instructions in this sub-format all encode a delay-slot mode, allowing an optional delay-slot instruction to be specified. For information about delay slot mode encoding, see Table 8-13.

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.

9.2.3.1 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 8-10.

Figure 9-5 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.

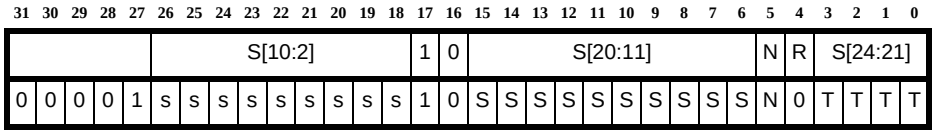
Syntax:

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

9.2.3.2 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 9-6 Branch and Link Unconditional Far Format



The reserved field, R, is ignored.

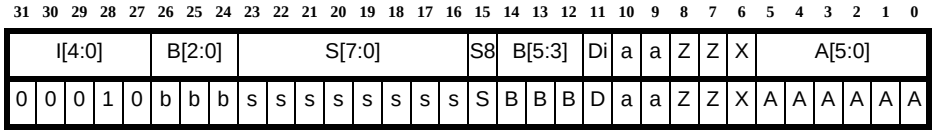
Syntax:

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

9.3 Load Register with Offset Format [F32_LD_OFFSET]

For information about encoding the Load instruction, see Table 8-11, Table 8-17, and Table 8-18.

Figure 9-7 Load Register with Offset Format



The program counter (PCL) is not permitted to be the destination of a load instruction. Values 0x3D and 0x3F in the destination register field, A, raise an Illegal Instruction exception.

Using incrementing addressing modes, with a long immediate value as the base register, is illegal, and raises an Illegal Instruction exception.

Syntax (note: not all combinations of modifier options are supported):

```
LD/LDL/LDDL<zz><.x><.aa>  a,[b,s9]
<.di>

LD/LDL/LDDL<zz><.x><.di>  a,[limm,s9]
```

If the s9 offset is omitted from the assembly code the assembler will set the s9 field to zero.

9.4 Store Register with Offset Format [F32_ST_OFFSET]

For information about encoding the Store instruction, see Table 8-12.

Figure 9-8 Store Register with Offset Format

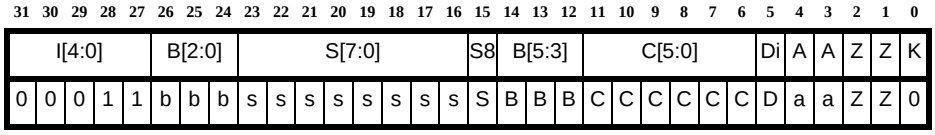
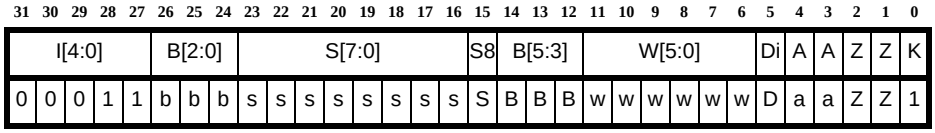


Figure 9-9 Store Immediate with Offset Format



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 (note: not all combinations of modifier options are supported):

ST/STL/STDL<zz><.aa><.di>	c,[b,s9]
ST/STL/STDL<zz><.di>	c,[limm, s9]
ST/STL/STDL<zz><.aa><.di>	limm,[b,s9]
ST<zz><.aa><.di>	w6,[b,s9]
ST<zz><.di>	w6,[limm,s9]

If the s9 offset is omitted from the assembly code the assembler will set the s9 field to zero.

9.5 General Operations Format [F32_GEN4]

The ARCv3 instruction set defines a collection of hierarchical formats that are used to encode the majority of instructions that are neither branches nor load-store operations. These typically define a monadic or dyadic operator, which is applies to one or two source operands. Many, but not all, of these operators write a result to their destination, and they may also have the capability to set arithmetic status flags. Certain sub-formats allow a condition (also known as a predicate) to be specified, allowing the instruction to be executed only when the value of that condition is true.

9.5.1 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 9-1 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 9-1 Hierarchical Sub-formats of the F32_GEN4 Format

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

9.5.2 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 9-2.

Table 9-2 Operand Sub-format Indicators

Format Name	P[1:0]	M	Operand Format Description
REG_REG	00	N/A	All operand fields specify registers

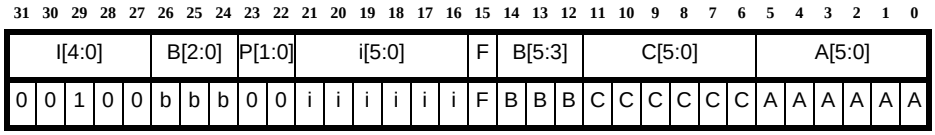
Table 9-2 Operand Sub-format Indicators (Continued)

Format Name	P[1:0]	M	Operand Format Description
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

9.5.3 Syntax and Encoding of Instructions in General Operations Formats

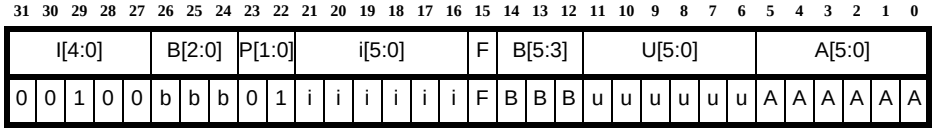
9.5.3.1 General Operations Register-register Format

Figure 9-10 General Operations Register-Register Format



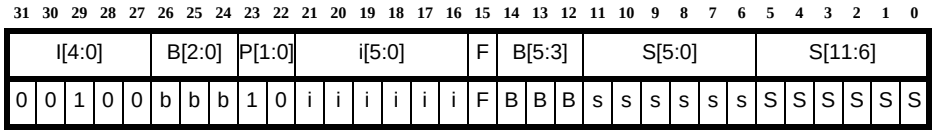
9.5.3.2 General Operations Register with Unsigned 6-bit Immediate

Figure 9-11 General Operations Register with Unsigned 16-bit Immediate Format



9.5.3.3 General Operations Register with Signed 12-bit Immediate

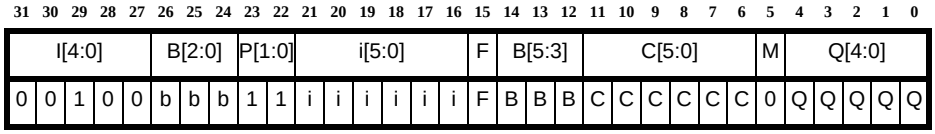
Figure 9-12 General Operations Register with Signed 12-bit Immediate Format



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.

9.5.3.4 General Operations Conditional Register

Figure 9-13 General Operations Conditional Register Format

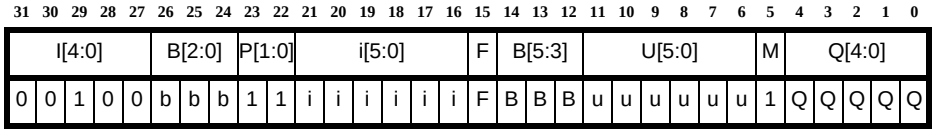


A value of 0x2F in the first operator field, i, indicates a SOP or ZOP format instruction, which is invalid for this operand mode and raises an Illegal Instruction exception.

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.

9.5.3.5 General Operations Conditional Register with Unsigned 6-bit Immediate

Figure 9-14 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.

9.5.4 ARC64 Extension Instruction Format [F32_GEN_OP64]

The extension instruction format F32_GEN_OP64 contains all of the instructions that are enabled by the ARC64_ISA configuration option. As this format uses the same major opcode (0x0B) as F16_JLI_EI, it is present only in ISA profiles that do not also contain the F16_JLI_EI format. The F32_GEN_OP64 format is subdivided into DOP, SOP and ZOP sub-formats.

9.5.4.1 Dual-operand Instructions, F32_GEN4

The DOP opcode assignments for F32_GEN4 are shown below in Table 9-3. Subsequent sections contain similar tables for SOP and ZOP format instructions, and for other top-level formats. Each cell in these table represents a unique opcode slot that may be occupied by at most one instruction. Each table has eight columns, representing the eight value of the most significant three bits of the instruction opcode i[5:3]. Each table has eight pairs of rows, with each pair representing one of the eight values of the least significant three bits of the instruction opcode i[5:3]. The first row of each pair encodes instructions that have the <.F> bit set to 0, and the second row of the pair encodes instructions for which the <.F> bit is 1.

Any attempt to execute an instruction with an opcode marked Illegal in one of the opcode tables, will raise an Illegal Instruction Exception.

A Reserved cell is one that is not allocated to an instruction, although attempting to execute the instruction with that opcode will not result in an exception. These cases indicate that the <.F> bit is ignored during the decoding of the instruction that is positioned directly above the Reserved cell.

Table 9-3 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

Table 9-3 Opcode assignment for DOP instructions in major op-code 0x04 (F32_GEN4)

000	0	ADD	MAX	BCLR	SUB2	J	Reserved	LD	SETEQ
	1	ADD.F	MAX.F	BCLR.F	SUB2.F	Reserved	Illegal		SETEQ.F
001	0	ADC	MIN	Illegal	SUB3	J.D	FLAG		SETNE
	1	ADC.F	MIN.F	BTST	SUB3.F	Reserved	KFLAG		SETNE.F
001	0	ADC	MIN	VMIN2	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	ENTER		SETLE
	1	BIC.F	RSUB.F	ADD3.F	MPYW.F	Reserved	LEAVE		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 9-4 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**	i[5:0]	R	R	C[5:0]	Reserved																							
J<cc>.D	**0x21**			**0 1**				U[5:0]																							
JL<cc>	**0x22**			**1 0**				S[5:0]	S[11:6]																							
JL<cc>.D	**0x23**			**1 1**				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																							
MOV<.cc><.F>	**0x0A**	0x4	B[2:0]	**0 0**	i[5:0]	F	B[5:3]	C[5:0]	Reserved																							
BTST<.cc>	**0x11 (F=1)**			**0 1**				U[5:0]																							
TST<.cc>	**0x0B (F=1)**			**1 0**				S[5:0]	S[11:6]																							
				1 1				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**	i[5:0]	F	R	C[5:0]	Reserved																							
KFLAG<.cc>	**0x29 (F=1)**			**0 1**				U[5:0]																							
				1 0				S[5:0]	S[11:6]																							
				1 1				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**	i[5:0]	R	B[5:3]	C[5:0]	Reserved																							
CMP<.cc>	**0x0C**			**0 1**		1		U[5:0]																							
RCMP<.cc>	**0x0D**			**1 0**				S[5:0]	S[11:6]																							
				1 1				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**	i[5:0]	R	B[5:3]	C[5:0]	Reserved																							
SR	**0x2B**			**0 1**				U[5:0]																							
LDI	**0x26**			**1 0**				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																							

9.5.5 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 9-5 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	LLOCKL	Illegal	Illegal	Illegal	Illegal	Illegal
	1	LSR.F	NOT.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	ROR	RLC	SCONDL	Illegal	Illegal	Illegal	Illegal	Illegal
	1	ROR.F	RLC.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	RRC	EX		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
	1	SEXB.F	ROL.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	SEXW		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	

9.5.6 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 9-6 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 9-6 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

9.5.7 Dual-operand Instructions, F32_GEN_OP64

The majority of 64-bit operations are encoded in the DOP sub-format of F32_GEN_OP64. These operations effectively replicate the operations available for 32-bit data, although with some exceptions and some additions.

Table 9-7 Opcode assignment for DOP instructions in major op-code 0x0B (F32_GEN_OP64)

Opcode		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ADDL	MAXL	BCLRL	SUB2L	ASLL	REML	MPYL	SETEQL
	1	ADDL.F	MAXL.F	BCLRL.F	SUB2L.F	ASLL.F	REML.F	VPACK4HL	SETEQL.F
001	0	ADCL	MINL	Illegal	SUB3L	LSRL	REMUL	MPYML	SETNEL
	1	ADCL.F	MINL.F	BTSTL	SUB3L.F	LSRL.F	REMUL.F	VPACK4HM	SETNEL.F
010	0	SUBL	MOVL	BXORL	Illegal	ASRL	LRL	MPYMUL	SETLTL
	1	SUBL.F	MOVL.F	BXORL.F	Illegal	ASRL.F	Illegal	VPACK2WL	SETLTL.F
011	0	SBCL	MOVHL	BMSKL	Illegal	Illegal	SRL	MPYMSUL	SETGEL
	1	SBCL.F	TSTL	BMSKL.F	Illegal	Illegal	Illegal	VPACK2WH	SETGEL.F
100	0	ANDL	Illegal	ADD1L	Illegal	DIVL	BMSKNL	Illegal	SETLOL
	1	ANDL.F	CMPL	ADD1L.F	Illegal	DIVL.F	BMSKNL.F	Illegal	SETLOL.F
101	0	ORL	Illegal	ADD2L	Illegal	DIVUL	XBFUL	Illegal	SETHSL
	1	ORL.F	RCMPL	ADD2L.F	Illegal	DIVUL.F	XBFUL.F	Illegal	SETHSL.F
110	0	BICL	RSUBL	ADD3L	Illegal	Illegal	ADDHL	Illegal	SETLEL
	1	BICL.F	RSUBL.F	ADD3L.F	Illegal	Illegal	Illegal	Illegal	SETLEL.F
111	0	XORL	BSETL	SUB1L	Illegal	AEXL	SOP_FMT	Illegal	SETGTL
	1	XORL.F	BSETL.F	SUB1L.F	Illegal	Illegal		Illegal	SETGTL.F

9.5.8 Single-operand Instructions, F32_GEN_OP64

Table 9-8 Opcode assignment for SOP instructions in major op-code 0x0B (F32_GEN_OP64)

SOP Code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ASLL	Illegal	LLOCKL	Illegal	SWAPL	Illegal	ATLDL.ADD	Illegal
	1	ASLL.F	Illegal	LLOCKL.AQ	Illegal	SWAPL.F	Illegal	ATLDL.ADD.AQ.RL	Illegal
001	0	ASRL	ABSL	SCONDL	Illegal	NORML	SWAPEL	ATLDL.OR	Illegal
	1	ASRL.F	ABSL.F	SCONDL.RL	Illegal	NORML.F	SWAPEL.F	ATLDL.OR.AQ.RL	Illegal
010	0	LSRL	NOTL	FFSL	Illegal	Illegal	Illegal	ATLDL.AND	Illegal
	1	LSRL.F	NOTL.F	FFSL.F	Illegal	Illegal	Illegal	ATLDL.AND.AQ.RL	Illegal
011	0	Illegal	Illegal	FLSL	Illegal	Illegal	Illegal	ATLDL.XOR	Illegal
	1	Illegal	Illegal	FLSL.F	Illegal	Illegal	Illegal	ATLDL.XOR.AQ.RL	Illegal
100	0	Illegal	EXL	Illegal	Illegal	Illegal	Illegal	ATLDL.MINU	Illegal
	1	Illegal	EXL.AQ.RL	Illegal	Illegal	Illegal	Illegal	ATLDL.MINU.AQ.RL	Illegal
101	0	SEXBL	Illegal	Illegal	Illegal	Illegal	Illegal	ATLDL.MAXU	Illegal
	1	SEXBL.F	Illegal	Illegal	Illegal	Illegal	Illegal	ATLDL.MAXU.AQ.RL	Illegal
110	0	SEXHL	Illegal	Illegal	Illegal	Illegal	Illegal	ATLDL.MIN	Illegal
	1	SEXHL.F	Illegal	Illegal	Illegal	Illegal	Illegal	ATLDL.MIN.AQ.RL	Illegal
111	0	SEXWL	Illegal	Illegal	Illegal	Illegal	Illegal	ATLDL.MAX	ZOP_FMT
	1	SEXWL.F	Illegal	Illegal	Illegal	Illegal	Illegal	ATLDL.MAX.AQ.RL	

9.5.9 Zero-operand Instructions, F32_GEN_OP64

There are no ZOP instruction defines in the F32_GEN_OP64 format, and hence all such encodings are illegal.

Table 9-9 Opcode assignment for ZOP instructions in major op-code 0x0B (F32_GEN_OP64)

ZOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	Illegal	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

Table 9-9 Opcode assignment for ZOP instructions in major op-code 0x0B (F32_GEN_OP64) (Continued)

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

9.5.10 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.

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

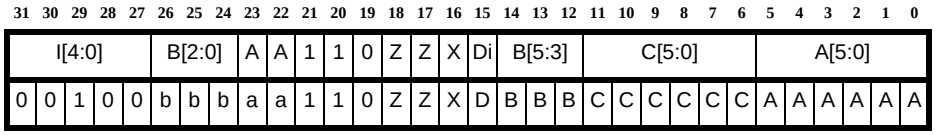
The jump (Jcc) and jump-and link (JLcc) instructions are specially encoded in major opcode 0x04 in which the B field is reserved and must be set to 0x0. Any value in the B field is ignored by the processor. The destination register, A field, must also be set to 0x0 when the operand mode, P, is 0x0 or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

9.5.12 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 8-17 and Table 8-18.

Figure 9-15 Load Register-Register Format



The program counter (PCL) is not permitted to be the destination of a load instruction. Values, 0x3D and 0x3F, in the destination register field, A, raise an Illegal Instruction exception.

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 value as the base register is illegal raises an Illegal Instruction exception.

9.6 F32_EXT5

9.6.1 Dual-operand Instructions, F32_EXT5

Table 9-10 Opcode assignment for DOP instructions in major op-code 0x05 (F32_EXT5)

opcode		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ASL multiple	REM	DMPYH	MPYD	Illegal	Illegal	QMPYH	VADD4H
	1	ASL multiple.F	REM.F	DMPY.F	MPYD.F	Reserved	Illegal	QMPYH.F	VMIN2
001	0	LSR multiple	REMU	DMPYHU	MPYDU	Illegal	VPACK2HL	QMPYHU	VSUB4H
	1	LSR multiple.F	REMU.F	DMPYHU.F	MPYDU.F	Illegal	VPACK2HM	QMYPHU.F	VMAX2
010	0	ASR multiple	Illegal	DMACH	MACD	Illegal	Illegal	DMPYWH	VADDSUB4H
	1	ASR multiple.F	Illegal	DMACH.F	MACD.F	Illegal	Illegal	DMPYWH.F	Illegal
011	0	ROR multiple	Illegal	DMACHU	MACDU	Illegal	Illegal	DMPYWHU	VSUBADD4H
	1	ROR multiple.F	Illegal	DMACHU.F	MACDU.F	Illegal	Illegal	DMPYWHU.F	Illegal
100	0	DIV	Illegal	VADD2H	VMPY2H	Illegal	Illegal	QMACH	VADD2
	1	DIV.F	Illegal	Illegal	Illegal	Illegal	Illegal	QMACH.F	Illegal
101	0	DIVU	Illegal	VSUB2H	VMPY2HU	Illegal	Illegal	QMACHU	VSUB2
	1	DIVU.F	Illegal	Illegal	Illegal	Illegal	Illegal	QMACHU.F	Reserved
110	0	Illegal	MAC	VADDSUB2H	Illegal	Illegal	Illegal	DMACWH	VADDSUB
	1	Illegal	MAC.F	Illegal	Illegal	Illegal	Reserved	DMACWH.F	Illegal
111	0	Illegal	MACU	VSUBADD2H	Illegal	Illegal	SOP_FMT	DMACWHU	VSUBADD
	1	Illegal	MACU.F	Illegal	Illegal	Illegal		DMACWHU.F	Illegal

9.6.2 Single-operand Instructions, F32_EXT5

Table 9-11 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	Illegal	Illegal	Illegal	Illegal	Illegal
		SWAP.F	NORMH.F	ROL8.F	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	NORM	SWAPE	ROR8	Illegal	Illegal	Illegal	Illegal	Illegal
	1	NORM.F	SWAPE.F	ROR8.F	Illegal	Illegal	Illegal	Illegal	Illegal
010	0	Illegal	LSL16	FFS	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	LSL16.F	FFS.F	Illegal	Illegal	Illegal	Illegal	Illegal
011	0	Illegal	LSR16	FLS	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	LSR16.F	FLS.F	Illegal	Illegal	Illegal	Illegal	Illegal
100	0	Illegal	ASR16	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	ASR16.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
101	0	Illegal	ASR8	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	ASR8.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
110	0	Illegal	LSR8	Illegal	Illegal	Illegal	Reserved	Illegal	Illegal
	1	Illegal	LSR8.F	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
111	0	Illegal	LSL8	Illegal	Illegal	Illegal	Reserved	Illegal	ZOP_FMT
	1	Illegal	LSL8.F	Illegal	Illegal	Illegal	Illegal	Illegal	

9.6.3 Zero-operand Instructions, F32_EXT5

Table 9-12 Opcode assignment for ZOP instructions in major op-code 0x05 (F32_EXT5)

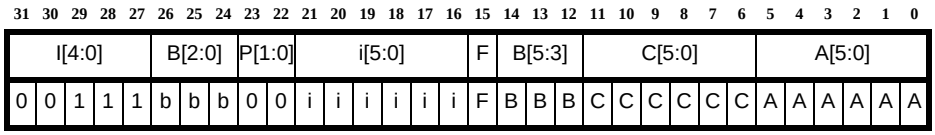
ZOP code		i[5:3]							
i[2:0]	.F	000	001	010	011	100	101	110	111
000	0	ASLACC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
001	0	ASLSACC	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
	1	Illegal	Illegal	Illegal	Illegal	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

9.7 APEX Extension Instruction Format [F32_APEX]

The F32_APEXtop-level format, assigned to major opcode 0x07, is available for use by User Extension Instructions defined using the APEX extension mechanism.

Figure 9-16 using major opcode 0x07 as an example to show the syntax of op<.f> a,b,c encoding.

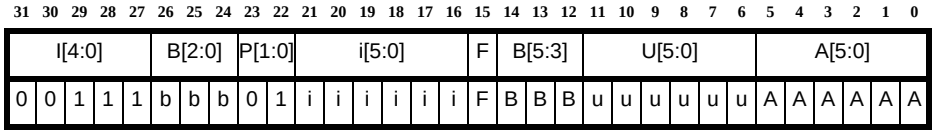
Figure 9-16 Extension ALU Operation, register-register



9.7.1 Extension ALU Operation, Register with Unsigned 6-bit Immediate

Figure 9-17 shows the APEX encoding syntax of op<.f> a,b,u6 encoding.

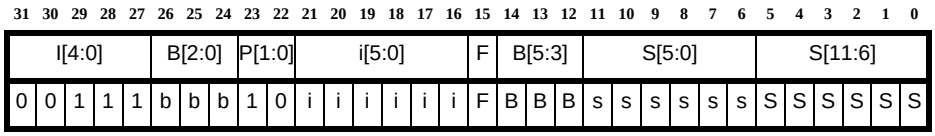
Figure 9-17 Extension ALU Operation, register with unsigned 6-bit immediate



9.7.2 Extension ALU Operation, Register with Signed 12-bit Immediate

Using major opcode 0x05 as an example, Figure 9-18 shows the syntax of op<.f> b,b,s12 encoding.

Figure 9-18 Extension ALU Operation, register with signed 12-bit immediate

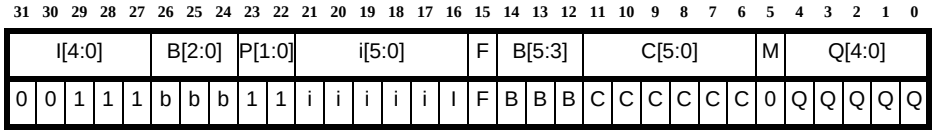


A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and raises an Illegal Instruction exception.

9.7.3 Extension ALU Operation, Conditional Register

Figure 9-19 shows the syntax of op<.cc><.f> b,b,c that is encoded.

Figure 9-19 Extension ALU Operation, conditional register



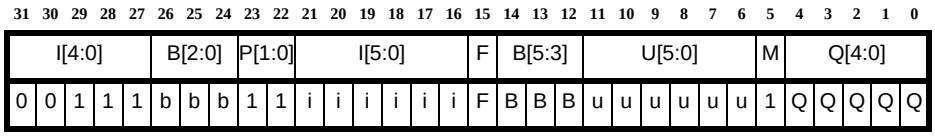
A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and raises an Illegal Instruction exception.

Values 0x10 to 0x1F in the condition code field, Q, raises an Illegal Instruction exception unless the Q field specifies a condition code defined as an extension condition.

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

Using major opcode 0x05 as an example, Figure 9-20 shows the syntax of op<.cc><.f> b,b,u6 encoding.

Figure 9-20 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.

9.7.5 FastMath Extension Pack Instructions

Table 9-13 lists the FastMath Pack Extension Instructions. If this extension is available with your processor, the instructions will be available in the *Part: FastMath Pack Extension* in the *ARCv3 ISA Programmer's Reference Manual*.

Table 9-13 List of FastMath Extension Pack Instructions

Instruction	Operation
FMP_ADDS	32-bit signed addition with result saturation
FMP_RNDH	Round and saturate a 32-bit signed 2's complement value to a 16-bit value
FMP_SATH	Saturate a 32-bit signed 2's complement integer to a 16-bit value
FMP_DIVF	Fractional division of Q31 numerator X, by Q31 divisor Y, where abs(X) < abs(Y) or X = -Y, and Y != 0.
FMP_DIVF15	Fractional division of Q15 numerator X, by Q15 divisor Y, where abs(X) < abs(Y) or X = -Y, and Y != 0.
FMP_RECIP	Computes the reciprocal of a Q31 fraction X, provided X != 1.
FMP_RECIP15	Computes the reciprocal of a Q15 fraction X, provided X != 1.
FMP_SQRTF	Computes the square root of the Q31 fractional argument X, when X >= 0.
FMP_SQRTF15	Computes the square root of the Q15 fractional argument X, when X >= 0.
FMP_COS	Computes the cosine function on the Q31 fractional input operand. The input operand represents an angle expressed in radians divided by Pi. The resulting cosine value is returned as the Q31 result.
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.

Table 9-13 List of FastMath Extension Pack Instructions

Instruction	Operation
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).

9.8 ARC64 Extension Instruction Format [F32_GEN_OP64]

The extension instruction format F32_GEN_OP64 contains all of the instructions that are enabled by the ARC64_ISA configuration option. As this format uses the same major opcode (0x0B) as F16_JLI_EI, it is present only in ISA profiles that do not also contain the F16_JLI_EI format. The F32_GEN_OP64 format is subdivided into DOP, SOP and ZOP sub-formats in exactly the same way as the F32_EXT5 and F32_EXT6 formats.

10
16-bit Instruction Formats Reference

This chapter explains the 16-bit encoding formats for ARCv3 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 8-6 illustrates all of the 16-bit top-level formats of the ARCv3 ISA in a concise form, and may be used as a reference. The list of syntax conventions is shown in Table 8-19, and section "Encoding Notation" on page 230 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 [F16_COMPACT0]
- Compact MOVL [F16_MOVL]
- Compact Load/Add/Sub [F16_LD_ADD_SUB]
- Compact Load/Store [F16_LD_ST_JLI]
- Load /Add Register-Register [F16_LD_ADD_RR]
- 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_OP64]
- Load/Add GP-Relative [F16_GP_OP64]
- Load PCL-Relative [F16_PCL_LD]
- Move Immediate [F16_MV_IMM]
- Branch on Compare Register with Zero [F16_BCC_REG]
- Branch Conditionally [F16_BCC]
- Branch and Link Unconditionally [F16_BL]
- Compact Long-jump and Long-call Instructions

Each section provides details of the sub-formats, and describes how instructions and their operands are encoded in those sub-formats.

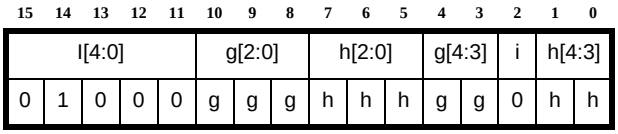
For a summary of the instructions discussed in this chapter, and an explanation of the functions they perform, see Chapter 8, "Instruction Set Summary".

For full details on the available encodings and syntax for each instruction, see the relevant page for each instructions in Chapter 11, "Instruction Set Details".

10.1 Compact Move [F16_COMPACT0]

Extension Group: BASELINE

Figure 10-1 Compact Move Format



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.

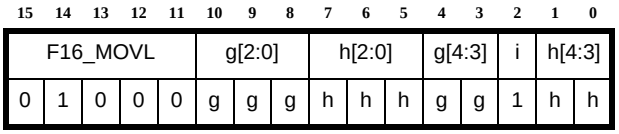
Syntax:

MOV_S

10.2 Compact MOVL [F16_MOVL]

Extension Group: BASELINE

Figure 10-2 Compact Move Format



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.

This instruction format is available only in the ARC64-based processors.

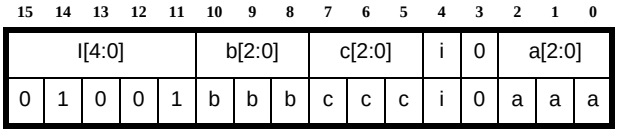
Syntax:

```
MOVL_S    g,h
```

10.3 Compact Load/Add/Sub [F16_LD_ADD_SUB]

Extension Group: CODE_DENSITY

Figure 10-3 Compact Load and Subtract Format

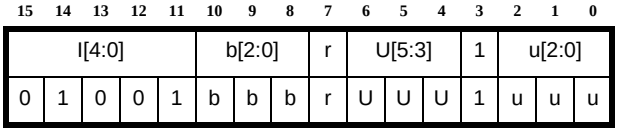


The first sub-format of opcode 0x09 encodes additional LD_S.AS and SUB_S instructions in a compact format to improve code density.

Syntax:

```
LD_S.AS     a, [b, c]
SUB_S       a, b, c
```

Figure 10-4 Compact Add Format



The second sub-format of opcode 0x09 encodes additional ADD_S instructions, with R0 or R1 as their destination register, in a compact form to improve code density.

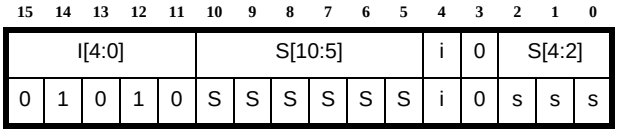
Syntax:

```
ADD_S       r, b, u6    r can be R0 or R1
```

10.4 Compact Load/Store [F16_LD_ST_JLI]

Extension Group: CODE_DENSITY

Figure 10-5 Compact Load and Store Format

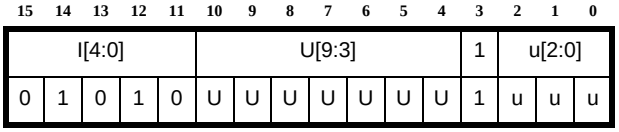


The first sub-format in opcode 0x0A encodes two additional GP-relative load/store instructions for use when accessing the most frequently occurring small data objects.

Syntax:

```
LD_S    R1, [GP, s11]
ST_S    R0, [GP, s11]
```

Figure 10-6 Compact Load Indexed Format

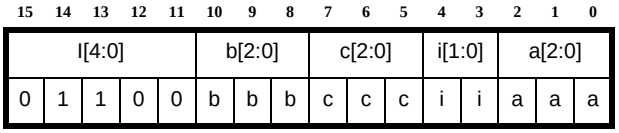


The second sub-format in opcode 0x0A encodes the JLI_S instruction.

```
JLI_S       u10
```

10.5 Load /Add Register-Register [F16_LD_ADD_RR]

Figure 10-7 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 10-1 16-Bit, LD / ADD Register-Register

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	LD_S	a ⟵ mem[b + c].l	Load 32-bit word (reg.+reg.)
0x01	LDB_S	a ⟵ mem[b + c].b	Load unsigned byte (reg.+reg.)
0x02	LDH_S	a ⟵ mem[b + c].w	Load unsigned 16-bit half-word (reg.+reg.)
0x03	ADD_S	a ⟵ b + c	Add

10.6 Dual Register Operations [F16_OP_HREG]

Figure 10-8 Dual Register with a 3-bit b Register Format

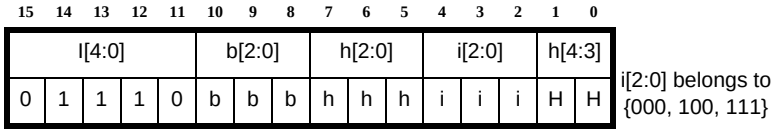


Figure 10-9 Dual Register with a 3-bit Biased Signed Integer Format

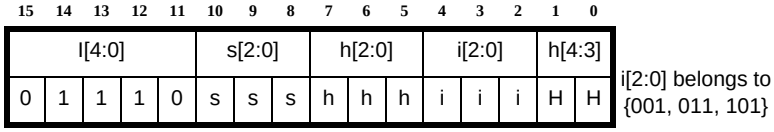
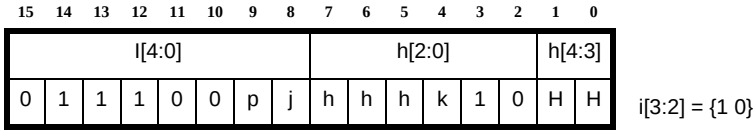


Figure 10-10 Dual Register with a Implicit LIMM Instructions



Format 0x0E encodes eight distinct operators, each with two register operands. The processor can determine if an instruction has a LIMM as an implied source operand of format 0x0E. Given the most-significant 16 bits of any instruction, the presence of an implicit LIMM due to these four new instructions can be computed thus:

```
implicit_limm = (inst[15:10] == 6'b011100) && (inst[3:2] == 2'b10);
```

when inst[3:2] = 2'b10, the 'p' bit selects whether the first source operand is the same as 'h' or is PCL as shown in Table 10-2.

Table 10-2 Interpretation of the p bit in the ARC64 F16_OP_HREG format

P	First source operand (rs1)
0	'h' register (r0 – r31)
1	PCL

Table 10-3 shows when inst[3:2] = 2'b10, the two opcode bits {j, k} encode the four operations.

Table 10-3 Interpretation of the {j, i} sub-opcode bits in the ARC64 F16_OP_HREG format

j k	Operation	LIMM Extension
00	MOVHL_S h, LIMM	NA
01	ORL_S h, rs1, LIMM	Zero-extended
10	ADDHL_S h, rs1, LIMM	NA

Table 10-3 Interpretation of the {j, i} sub-opcode bits in the ARC64 F16_OP_HREG format

j k	Operation	LIMM Extension
11	ADDL_S h, rs1, LIMM	Sign-extended

A logical operation (MOVL or ORL) is implied when j = 0, and an ADD operation is implied when j = 1. Conversely, a *high* variant of the operation (that is, MOVHL or ADDHL) is implied when k = 0, whereas a normal operation (ORL or ADDL) is implied when k = 1.

When using these compact versions of MOVHL, ORL, ADDHL and ADDL, it is possible to encode each of the two code sequences in use-case (a) and (b) with just 12 bytes (8 bytes of literal value and 4 bytes of instruction opcodes).

The F16_OP_HREG format (0x0E) contains sufficient unused encodings to permit compact encodings of the most common 64-bit instructions needed to introduce 64-bit literals into a program. There are four specific 64-bit instructions that are expected to be commonly used to either assign a 64-bit literal to a GPR or assign PCL + 64-bit literal to a GPR.

These use-cases are listed in:

- "Assign a 64-bit Literal to a GPR" on page 311
- "Assign PCL + 64-bit literal to a GPR" on page 311

10.6.1 Assign a 64-bit Literal to a GPR

There are several possible ways to assign a 64-bit literal to a GPR. These include loading it from data memory, loading it from program memory (using a PC-relative addressing mode), or constructing the literal from two 32-bit LIMM values using a pair of instructions. The instruction sequence used for this latter option is shown in Figure 10-11.

Figure 10-11 Code Sequence to Construct a 64-bit Literal using Two LIMMs

```
MOVHL rA, (literal64 >> 32)
ORL rA, rA, (literal64 & 0xffffffff)
```

The MOVHL instructions places the upper 32 bits of literal64 in the upper 32 bits of rA and clears the lower 32 bits of rA. The ORL instruction places the lower 32 bits of literal64 into the lower 32 bits of rA.

10.6.2 Assign PCL + 64-bit literal to a GPR

There are several possible ways to add a 64-bit literal to a GPR. These include first loading it into a GPR using the sequence given in Figure 10-12, or adding the literal in two halves using two 64-bit add operations and two 32-bit LIMM values. The instruction sequence used for this latter option is shown Figure 10-12.

Figure 10-12 Code sequence to add a 64-bit literal to a GPR using two LIMMs

```
ADDHL rA, PCL, (literal64 >> 32)+((literal64 >> 31) & 1)
ADDL rA, rA, (literal64 & 0xffffffff)
```

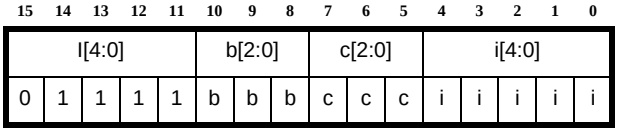
The literal operand of the first ADDHL instruction contains a sign-correcting term to add 1 at bit-position 32 of rA if bit 31 of literal64 is 1. In this case the lower 32 bits of literal64 represent a negative 32-bit integer, which will be signed extended by the second ADDL instruction – effectively subtracting 1 at bit-position 32 of rA.

It is expected that an assembler for ARC64 will provide a pseudo-instruction that will expand into this sequence during assembly.

10.7 General Register Format Instructions [F16_GEN_OP]

10.7.1 DOP-format General Operations

Figure 10-13 Compact General Operations Register-Register Format



Syntax:

op_S b,b,c

op_S b,c

Table 10-4 DOP_format 16-Bit General Operations

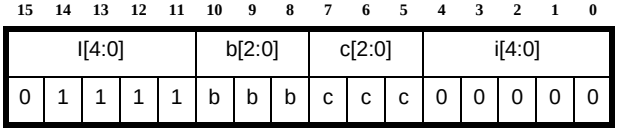
Sub-opcode i field (5 bits)	Instruction	Operation	Description
0x00	SOPs	c field is sub-opcode2	See Table 10-5.
0x01	ADDL_S b, b, c		64-bit addition
0x02	SUB_S	b ⟵ b – c	Subtract
0x03	SUBL_S b, b, c		64-bit subtraction
0x04	AND_S	b ⟵ b and c	Logical bitwise AND
0x05	OR_S	b ⟵ b or c	Logical bitwise OR
0x06	BIC_S	b ⟵ b and not c	Logical bitwise AND with invert
0x07	XOR_S	b ⟵ b exclusive-or c	Logical bitwise exclusive-OR
0x08	ANDL_S b, b, c		64-bit bitwise AND
0x09	MPYW_S	b ⟵ b * c	Multiply 16-bit half-words
0x0A	MPYUW_S	b ⟵ b * c (unsigned)	Multiply unsigned 16-bit half-words

Table 10-4 DOP_format 16-Bit General Operations (Continued)

Sub-opcode i field (5 bits)	Instruction	Operation	Description
0x0B	TST_S	b and c	Test
0x0C	MPY_S	b ⟵ b * c	32 X 32 Multiply (least-significant half)
0x0D	SEXB_S	b ⟵ sexb c	Sign extend byte
0x0E	SEXH_S	b ⟵ sexw c	Sign extend 16-bit half-word
0x0F	EXTB_S	b ⟵ extb c	Zero extend byte
0x10	EXTH_S	b ⟵ extw c	Zero extend 16-bit half-word
0x11	ABS_S	b ⟵ abs c	Absolute
0x12	NOT_S	b ⟵ not c	Logical NOT
0x13	NEG_S	b ⟵ neg c	Negate
0x14	ADD1_S	b ⟵ b + (c << 1)	Add with left shift by 1
0x15	ADD2_S	b ⟵ b + (c << 2)	Add with left shift by 2
0x16	ADD3_S	b ⟵ b + (c << 3)	Add with left shift by 3
0x17	ORL_S b, b, c		64-bit bitwise OR
0x18	ASL_S	b ⟵ b asl c	Multiple arithmetic shift left
0x19	LSR_S	b ⟵ b lsr c	Multiple logical shift right
0x1A	ASR_S	b ⟵ b asr c	Multiple arithmetic shift right
0x1B	ASL_S	b ⟵ c + c	Arithmetic shift left by one
0x1C	ASR_S	b ⟵ c asr 1	Arithmetic shift right by one
0x1D	LSR_S	b ⟵ c lsr 1	Logical shift right by one
0x1E	TRAP_S	Trap	Raise Exception
0x1F	BRK_S	Break	Break (Encoding is 0x7FFF)

10.7.2 SOP-format 16-bit General Operations

Figure 10-14 Compact Single Operand, Jumps, Special Formats



Syntax:

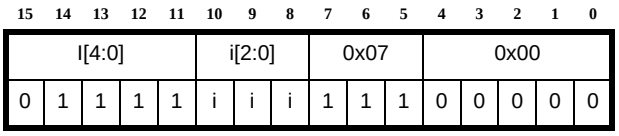
```
op_S        b
op_S        b,b
J_S<.d>     [b]
JL_S<.d>    [b]
SUB_S.ne    b,b,b
```

Table 10-5 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 10-6.

10.7.3 ZOP-format 16-bit General Operations

Figure 10-15 Compact Zero Operand Instructions



Syntax:

op_S

NOP_S

UNIMP_S

J_S<.d> [blink]

JEQ_S [blink]

JNE_S [blink]

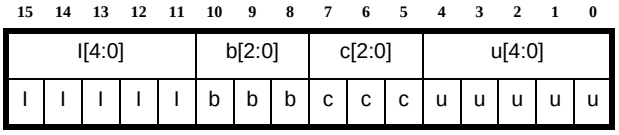
Table 10-6 16-Bit Zero Operand Instructions

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

10.8 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 10-16 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]

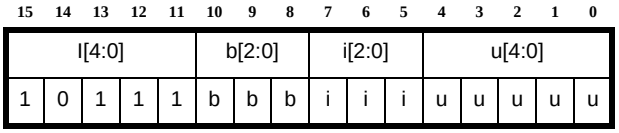
STH_S c, [b, u6] *(u6 must be 16-bit aligned)*

Table 10-7 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 ⟵ mem[b + u7].l	Load 32-bit word
F16_LD_BYTE	0x11	LDB_S	c ⟵ mem[b + u5].b	Load unsigned byte
F16_LD_HALF	0x12	LDH_S	c ⟵ mem[b + u6].w	Load unsigned 16-bit half-word
F16_LDX_HALF	0x13	LDH_S.X	c ⟵ mem[b + u6].wx	Load signed 16-bit half-word
F16_ST_WORD	0x14	ST_S	mem[b + u7].l ⟵ c	Store 32-bit word
F16_ST_BYTE	0x15	STB_S	mem[b + u5].b ⟵ c	Store unsigned byte
F16_ST_HALF	0x16	STH_S	mem[b + u6].w ⟵ c	Store unsigned 16-bit half-word

10.9 Shift/Subtract/Bit Immediate [F16_SH_SUB_BIT]

Figure 10-17 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 10-8 16-Bit Shift/SUB/Bit Immediate

Sub-opcode2 i field (3 bits)	Instruction	Operation	Description
0x00	ASL_S	b ⟵ b asl u5	Multiple arithmetic shift left
0x01	LSR_S	b ⟵ b lsr u5	Multiple logical shift right
0x02	ASR_S	b ⟵ b asr u5	Multiple arithmetic shift right
0x03	SUB_S	b ⟵ b – u5	Subtract
0x04	BSET_S	b ⟵ b or 1<<u5	Bit set
0x05	BCLR_S	b ⟵ b and not 1<<u5	Bit clear
0x06	BMSK_S	b ⟵ b and ((1<<(u5+1))-1)	Bit mask
0x07	BTST_S	b and 1<<u5	Bit test

10.10 Stack-based Operations [F16_SP_OP64]

Instruction group 0x18 contains a collection of 16-bit formats designed specifically to encode common stack-based operations in which the SP register (r28) in always an implied operand. These instructions compute at 64-bit precision.

Only 7 bits of the u9 operand [9:2] are stored in the instruction; the other 2 bits continue to be zero, as SP continues to be 4-byte aligned in the 64-bit ABI. Thus the maximum size of stack frame that can be created or destroyed by these SUB_S and ADD_S instructions is 512 bytes. For ARC64, arguments, automatic variables, and register spills, can now be twice the size. The additional SUB_S / ADD_S range for SP allows the same number (or more) stack pointer changes to be encoded in 16-bit formats.

Table 10-9 Opcodes for Stack-based Operations

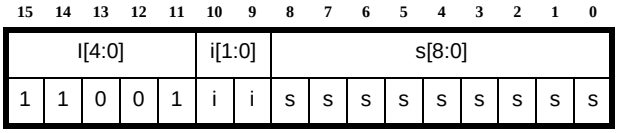
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Encoded Instructions	**I[4:0]**					**b[2:0]**			**i[2:0]**			**u[4:0]**				
LD_S b, [SP, u7]	1	1	0	0	0	b	b	b	0	0	0	u	u	u	u	u
LDB_S b, [SP, u7]	1	1	0	0	0	b	b	b	0	0	1	u	u	u	u	u
ST_S b, [SP, u7]	1	1	0	0	0	b	b	b	0	1	0	u	u	u	u	u
STB_S b, [SP, u7]	1	1	0	0	0	b	b	b	0	1	1	u	u	u	u	u
ADDL_S b, SP, u7	1	1	0	0	0	b	b	b	1	0	0	u	u	u	u	u
ADDL_S SP, SP, u9	1	1	0	0	0	U	U	0	1	0	1	u	u	u	u	u
SUBL_S SP, SP, u9	1	1	0	0	0	U	U	1	1	0	1	u	u	u	u	u
POPL_S b	1	1	0	0	0	b	b	b	1	1	0	0	B	B	B	1
POPDL_S b	1	1	0	0	0	b	b	b	1	1	0	1	B	B	B	1
PUSHL_S b	1	1	0	0	0	b	b	b	1	1	1	0	B	B	B	1
PUSHDL_S b	1	1	0	0	0	b	b	b	1	1	1	1	B	B	B	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
unused (Illegal Instruction)	1	1	0	0	0	1	-	-	1	1	1	-	-	-	-	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

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.

10.11 Load/Add GP-Relative [F16_GP_OP64]

Figure 10-18 Compact Load/Add GP-Relative Format



The offset (s[8:0]) is shifted accordingly to provide the appropriate data size alignment. This register is renamed as F16_GP_OP64 in ARC64.

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)*
ADDL_S	r0, GP, s11	*(64-bit aligned offset)*

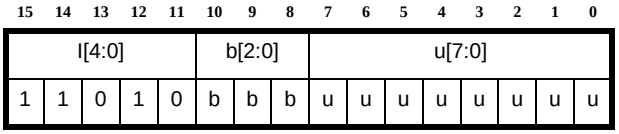
Table 10-10 16-Bit GP Relative Instructions

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	LD_S	r0 ⟵ mem[GP + s11].w	Load GP-relative (32-bit aligned) to r0
0x01	LDB_S	r0 ⟵ mem[GP + s9].b	Load unsigned byte GP-relative (8-bit aligned) to r0
0x02	LDH_S	r0 ⟵ mem[GP +s10].h	Load unsigned half-word GP-relative (16-bit aligned) to r0
0x03	ADDL_S	r0 ⟵ GP + s11	Add long GP to r0

10.12 Load PCL-Relative [F16_PCL_LD]

The offset (u[7:0]) is shifted accordingly to provide the appropriate 32-bit data size alignment.

Figure 10-19 Compact Load PCL Relative Format

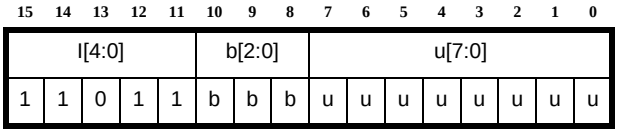


Syntax:

LD_S b, [PCL, u10] *(32-bit aligned offset)*

10.13 Move Immediate [F16_MV_IMM]

Figure 10-20 Compact Move Immediate Format



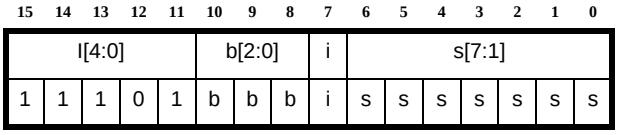
Syntax:

MOV_S b, u8

10.14 Branch on Compare Register with Zero [F16_BCC_REG]

BREQL_S and BRNEL_S perform comparisons on all 64 bits of their source register.

Figure 10-21 Compact Branch on Compare Register with Zero Format



Syntax:

BREQL_S b, 0, s8

BRNEL_S b, 0, s8

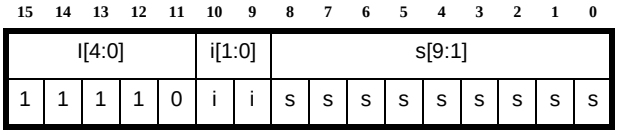
Table 10-11 16-Bit Branch on Compare

Sub-opcode i field (1 bit)	Instruction	Operation	Description
0x00	BREQL_S	if (b==0) PC = (PCL +s8)	Branch if register is zero
0x01	BRNEL_S	if (b!=0) PC = (PCL +s8)	Branch if register is non-zero

10.15 Branch Conditionally [F16_BCC]

The target address is 16-bit aligned.

Figure 10-22 Compact Branch Conditionally Format



Syntax:

```
B_S     s10
BEQ_S   s10
BNE_S   s10
```

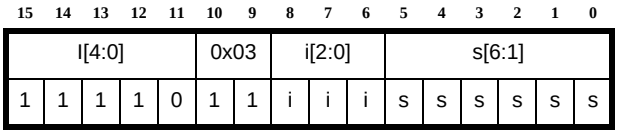
Table 10-12 16-Bit Branch, Branch Conditionally

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

10.15.1 Branch Conditionally with cc Field, 0x1E, [0x03, 0x00 – 0x07]

The target address is 16-bit aligned.

Figure 10-23 Branch Conditionally with cc Field Format



Syntax:

```
BGT_S  s7
BGE_S  s7
BLT_S  s7
BLE_S  s7
BHI_S  s7
BHS_S  s7
BLO_S  s7
BLS_S  s7
```

Table 10-13 16-Bit Branch Conditionally

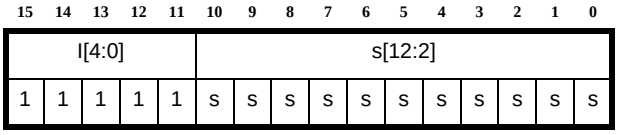
Sub-opcode i field (3 bits)	Instruction	Operation	Description
0x00	BGT_S	if ((N and V and /Z) or (/N and /V and /Z)) PC = (PCL+s7)	Branch if greater than
0x01	BGE_S	if ((N and V) or (/N and /V)) PC = (PCL+s7)	Branch if greater than or equal
0x02	BLT_S	if ((N and /V) or (/N and V)) PC = (PCL+s7)	Branch 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 8-9.

10.16 Branch and Link Unconditionally [F16_BL]

The target address can only target 32-bit aligned instructions.

Figure 10-24 Compact Branch and Link Unconditionally Format



Syntax:

```
BL_S  s13
```

10.17 Compact Long-jump and Long-call Instructions

Three new 16-bit encodings for J_S, JL_S and BL_S are included in the baseline ARC64 ISA to support long-jump and long-call. The key feature of these instructions is that they all have an implicit LIMM operand, which is combined with either PCL or one of the eight registers accessible by the 3-bit 'b' operand field of compact instructs to define the target address.

There are two new SOP encodings in the F16_GEN_OP format, one for J_S and one for JL_S, both of which have operands [b, LIMM]. In common with other F16_GEN_OP instructions, the 'b' operand register is encoded as a 3-bit field capable of specifying one of the eight registers r0, r1, r2, r3, r12, r13, r14, r15. For these two instructions, bits [63:32] of the jump target are provided by upper 32 bits of the 'b' operand, bits [31:1] of the jump target are provided by upper 31 bits of the LIMM, and bit 0 of the jump target is always 0 to enforce half-word alignment of the target.

There is also a new ZOP encoding in the F16_GEN_OP format for BL_S, which also has a LIMM operand. In this case the LIMM is sign-extended, left-shifted by 2 bit positions, and added to PCL to form the branch target.

Figure 10-25 Compact Long-jump and Long-call Format J_S [LIMM]

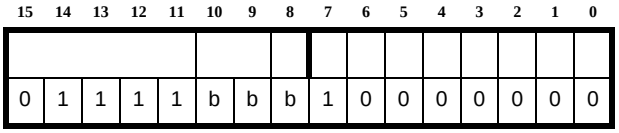


Figure 10-26 Compact Long-jump and Long-call Format JL_S [LIMM]

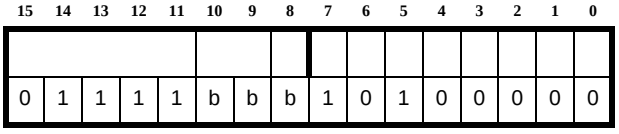
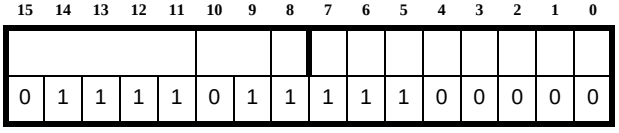


Figure 10-27 Compact Long-jump and Long-call Format BL_S [LIMM]



The target address calculations for the two operand formats are formally defined in Table 10-14.

Table 10-14 Jump Target Address Calculations for Long Jump/Call Instructions

Long Call/Jump Instructions	Jump Target Address Calculation
`J_S [b,LIMM]`	{ Rb[63:32], LIMM[31:1], 1'b0 }
`JL_S [b,LIMM]`	
`BL_S LIMM`	PCL + { {30{LIMM[31]}}, LIMM[31:0], 2'b00 }

The BL_S instruction with a LIMM operand therefore provides a signed 34-bit PC relative byte offset, giving a relative target address range of +/- 8GB.

As there is also a BL_S instruction that has a signed 13-bit offset, the assembler syntax of a BL_S that requires a signed 34-bit offset is differentiated from the signed 13-bit offset format by using a label suffix of "@s3", thus:

```
BL_S    label@s34
```

The two new JL_S and J_S instructions provide a compact encoding for a long jump or long call to any literal address within the 64-bit address space using two 48-bit instructions each containing 32 bits of the 64-bit literal, as shown :

```
MOVHL_S b,target@hi
JL_S    [b,target@lo]
```

11 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 ARCv3 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 8-19 lists the syntax conventions. Table 8-7 and Table 8-8 list the encoding notation.

Table 11-1 summarizes the 32-bit format instructions alongside the 16-bit format instructions supported by the ARCv3 ISA.

11.1 Instruction List

Table 11-1 List of Instructions

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
ABS	Absolute value	ABS_S	Absolute value
ABSL	Absolute long value		
ADC	Add with carry		
ADCL	Add long with carry		
ADD	Add	ADD_S	Add

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
ADDL	Add long		
ADDHL	Add high long with 32-bit shift		
ADD1	Add with left shift by 1 bit	ADD1_S	Add with left shift by 1 bits
ADD1L	Add long with 1-bit left shift		
ADD2	Add with left shift by 2 bits	ADD2_S (see ADD2)	Add with left shift by 2 bits
ADD2L	Add long with 2-bit left shift		
ADD3	Add with left shift by 3 bits	ADD3_S (see ADD3)	Add with left shift by 3 bits
ADD3L	Add long with 3-bit left shift		
AEX	Swap contents of auxiliary register with a 32-bit core register		
AEXL	Swap contents of auxiliary register with a 64-bit core register		
AND	Logical AND	AND_S (see AND)	Logical AND
ANDL	Logical AND long		
ASL	Arithmetic Shift Left	ASL_S (see ASL)	Arithmetic Shift Left
ASLL	Arithmetic Shift Left long		
ASL Multiple	Multiple Arithmetic Shift Left		
ASLL Multiple	Multiple Arithmetic Shift Left Long		
ASR	Arithmetic Shift Right	ASR_S (see ASR)	Arithmetic Shift Right
ASRL	Arithmetic Shift Right long		
ASR multiple	Multiple Arithmetic Shift Right		
ASRL Multiple	Multiple Arithmetic Shift Right Long		
ASR16	Arithmetic Shift Right by 16		
ASR8	Arithmetic Shift Right by 8		
ATLD	Atomic memory operations		
ATLDL	Atomic memory operations		
B	Branch unconditionally	B_S	Branch unconditionally

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
BBIT0	Branch if bit equal to 0		
BBIT0L	Branch if bit equal to 0		
BBIT1	Branch if bit equal to 1		
BBIT1L	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)
BCLRL	Clear specified bit (to 0) long		
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
BICL	Bit-wise inverted AND long		
BLcc	Branch and Link	BL_S (see BLcc)	Branch and Link
BMSK	Bit Mask	BMSK_S (see BMSK)	Bit Mask
BMSKL	Bit mask long		
BMSKN	Bit Mask Negated		
BMSKNL	Bit mask negated long		
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)
BSETL	Set specified bit (to 1) long		
BTST	Test value of specified bit	BTST_S (see BTST)	Test value of specified bit
BTSTL	Test value of specified bit long		
BXOR	Bit XOR		
BXORL	Bit XOR long		
CLRI	Clear Interrupt Enable		

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
CMP	Compare	CMP_S (see CMP)	Compare
CMPL	Compare long		
DBNZ	Decrement register and branch if the resulting value is non-zero		
DIV	Signed integer divide		
DIVL	Signed integer divide long		
DIVU	Unsigned integer Divide		
DIVUL	Unsigned integer divide long		
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		
DMPYWHU	Dual unsigned 32x16 multiplication		
DSYNC	Synchronize instruction		
ENTER		ENTER_S	Function prolog Sequence
EX	Atomic Exchange		
EXL	Atomic Exchange long		
EXTB	Zero-extend byte to word	EXTB_S (see EXTB)	Zero-extend byte

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
EXTH	Zero-extend 16-bit half-word to word	EXTH_S (see EXTH)	Zero-extend 16-bit half-word
FFS	Find First Set		
FFSL	Find First Set		
FLAG	Set Flags		
FLS	Find Last Set		
FLSL	Find Last Set		
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		
LDB	Load byte from memory		
LDH	Load half-word from memory		
LD	Load word from memory		
LDL	Load long-word from memory		
LDDL	Load double long-word from memory		
		LEAVE	Function Epilog Sequence
		LEAVE_S	Function Epilog Sequence
LLOCK	Load locked		
LLOCKL	Load locked on long word (ARC-64 ABI)		
LR	Load from Auxiliary memory		
LRL	Load long from Auxiliary memory		
LSL16	Logical Shift Left 16		
LSL8	Logical Shift Left 8		
LSR	Logical Shift Right	LSR_S (see LSR)	Logical Shift Right
LSRL	Logical Shift Left long		

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
LSR16	Logical Shift Right 16		
LSR8	Logical Shift Right 8		
MAC	32x32 multiply and accumulate		
MACD	Signed 32x32 multiplication and accumulation		
MACDU	Unsigned 32x32 multiply and accumulate, double result		
MACU	Unsigned 32x32 multiply and accumulate		
MAX	Return Maximum		
MAXL	Return long Maximum		
MIN	Return Minimum		
MINL	Return long Minimum		
MOV	Move (copy) to register	MOV_S (see MOV)	Move (copy) to register
MOVL	Move (copy) to long register		
MOVHL	Move to 32-bit high part of long destination		
MPY, MPY_S	32 x 32 Signed Multiply (lsw)	MPY_S (see MPY, MPY_S)	32 x 32 Signed Multiply (lsw)
MPYL	Signed 64x64 multiply		
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)		
MPYML	64 x 64, upper 64 bits of product		
MPYMUL	64 x 64, unsigned x unsigned, upper 64 bits of product		
MPYMSUL	64 x 64, signed x unsigned, upper 64 bits of product		

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
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		
NORML	Normalize to 64 bits		
NOT	Logical bit inversion	NOT_S (see NOT)	Logical bit inversion
NOTL	Logical bit inversion long		
OR	Logical OR	OR_S (see OR)	Logical OR
ORL	Logical OR long		
		POPL_S	Restore register from stack
		POPDL_S	Restore register from stack
PREALLOC	Allocate cache line as a preparation for a store		
PREFETCH	Prefetch from memory		
PREFETCHW	Prefetch line from Memory with intention to write		
		PUSHL_S	Store register to the stack
		PUSHDL_S	Store register to the stack
QMACH	Quad 16x16 multiply and accumulate		
QMACHU	Quad unsigned 16x16 multiply and accumulate		
QMPYH	Quad 16x16 multiplication		

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
QMPYHU	Quad unsigned 16x16 multiplication		
RCMP	Reverse Compare		
RCMPL	Reverse Compare long		
REM	Signed Integer Remainder		
REML	Signed Integer Remainder long		
REMU	Unsigned Integer Remainder		
REMUL	Unsigned Integer Remainder long		
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		
RSUBL	Reverse Subtraction long		
RTIE	Return from Interrupt or Exception		
SBC	Subtract with carry		
SBCL	Subtract with carry long		
SCOND	Store conditionally		
SCONDL	Store conditionally on 64-bit data (ARC64 ABI)		
SETcc	Set conditional		
SETccL	Set conditional long		
SETI	Set Interrupt Enable		
SEXB	Sign-extend byte to word	SEXB_S (see SEXB)	Sign-extend byte
SEXBL	Sign-extend byte to long		
SEXHL	Sign-extend half-word to long		

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
SEXWL	Sign-extend word to long		
SLEEP	Put processor in sleep state		
SR	Store to Auxiliary memory		
SRL	Store long to Auxiliary memory		
addr = b + offset; *addr = c;	Store byte to memory		
STH	Store half-word to memory		
ST	Store word to memory		
STL	Store long-word to memory		
STDL	Store double long-word to memory		
SUBL	Subtract long		
SUB1	Subtract with left shift by 1 bit		
SUB1L	Subtract long with left shift by 1 bit		
SUB2	Subtract with left shift by 2 bits		
SUB2L	Subtract long with left shift by 3bit		
SUB3	Subtract with left shift by 3 bits		
SUB3L	Subtract long with left shift by 3bit		
SWAP	Swap 16-bit register halves		
SWAPL	Swap 32-bit register halves		
SWAPE	Swap byte ordering		
SWAPEL	Swap byte ordering long		
SWI	Software interrupt	SWI_S (see SWI)	Software interrupt
SYNC	Synchronize		
		TRAP_S	Trap to system call
TST	Test	TST_S (see TST)	Test
TSTL	Test long		

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
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		
VMPY2HU	Dual unsigned 16x16 vector multiplication		
VPACK4HL	Pack even half-words		
VPACK4HM	Pack odd half-words		
VPACK2WL	Pack even words		
VPACK2WM	Pack odd words		
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		
WAIT	Wait for Interrupt, Event, or Lock Flag Clear		
WEVT	Enter sleep state and wait on event		

Table 11-1 List of Instructions (Continued)

32-Bit Instruction Formats		16-Bit Instruction Formats	
Instruction	**Operation**	**Instruction**	**Operation**
WLFC	Enter Sleep State to reduce dynamic power during busy-waiting loops		
XBFU	Extract unsigned bit-field		
XBFUL	Extract unsigned bit-field long		
XOR	Logical Exclusive-OR	XOR_S (see XOR)	Logical Exclusive-OR
XORL	Logical Exclusive-OR long		
		UNIMP_S	Unimplemented Instruction

11.2 ARC Instructions

This section provides a complete definition for each ARCv3 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 8-19 lists the instruction syntax convention. Table 8-7 and Table 8-8 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 Table 8-7 and Table 8-8.

		Instruction Code
ABS<.f>	b,c	00100bbb00101111FBBBCCCCCC001001
ABS<.f>	b,u6	00100bbb01101111FBBBuuuuuu001001
ABS_S	b,c	01111bbbccc10001

ABSL

Function

Absolute long

Extension Group

Baseline

Operation

b = ABS(c);

Instruction Format

op b, c

Syntax Example

ABSL<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if src = 0x8000 0000_0000_0000
C	•	= MSB of src
V	•	= Set if src = 0x8000 0000_0000_0000

Description

Take the absolute value that is found in the 64-bit source operand (c) and place the result into the 64-bit 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.

Raise an Illegal Instruction exception if the processor is not executing.

Pseudo Code

```
/* ABSL */
alu = 0 - src
if src[63]==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_0000_0000 then 1 else 0
 STATUS32[C] = src[31]
 STATUS32[V] = if src==0x8000_0000_0000_0000 then 1 else 0
```

Assembly Code Example

```
ABSL r1,r2    ; Take the absolute value of r2 and write result into r1
```

Syntax and Encoding

		Instruction Code
ABSL<.f>	b,c	01011bbb00101111FBBBCCCCCC001001
ABSL<.f>	b,u6	01011bbb01101111FBBBuuuuuu001001

ADC

Function

Addition with Carry

Extension Group

Baseline

Operation

if (cc) a = b + c + 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 Table 8-7 and Table 8-8.

		Instruction Code
ADC<.f>	a,b,c	00100bbb00000001FBBBCCCCCCAAAAAA
ADC<.f>	a,b,u6	00100bbb01000001FBBBuuuuuuAAAAAA
ADC<.f>	b,b,s12	00100bbb10000001FBBBssssssSSSSSS
ADC<.cc><.f>	b,b,c	00100bbb11000001FBBBCCCCCC0QQQQQ
ADC<.cc><.f>	b,b,u6	00100bbb11000001FBBBuuuuuu1QQQQQ

ADCL

Function

Long Addition with Carry

Extension Group

Baseline

Operation

if (cc) a = b + c + carry

Instruction Format

op a, b, c

Syntax Example

ADCL<.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 64-bit source operand 1 (b), 64-bit source operand 2 (c), and the carry flag, and place the 64-bit 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                    /* ADCL */
 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

```
ADCL r1,r2,r3   ; Add r2 to r3 with carry and write result into r1
```

Syntax and Encoding

		Instruction Code
ADCL<.f>	a,b,c	01011bbb00000001FBBBCCCCCCAAAAAA
ADCL<.f>	a,b,u6	01011bbb01000001FBBBuuuuuuAAAAAA
ADCL<.f>	b,b,s12	01011bbb10000001FBBBssssssSSSSSS
ADCL<.cc><.f>	b,b,c	01011bbb11000001FBBBCCCCCC0QQQQQ
ADCL<.cc><.f>	b,b,u6	01011bbb11000001FBBBuuuuuu1QQQQQ

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.

For 16-bit encoded instructions that work on the stack pointer (SP) or global pointer (GP), the offset is aligned to 64-bit.

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 Table 8-7 and Table 8-8.

		Instruction Code
ADD<.f>	a,b,c	00100bbb00000000FBBBCCCCCCAAAAAA
ADD<.f>	a,b,u6	00100bbb01000000FBBBuuuuuuAAAAAA
ADD<.f>	b,b,s12	00100bbb10000000FBBBssssssSSSSSS
ADD<.cc><.f>	b,b,c	00100bbb11000000FBBBCCCCCC0QQQQQ
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	01110bbb110000011
ADD_S	0,limm,s3	01110sss110001011
ADD_S	b,sp,u7	Synonymous with ADDL_S b, sp, u7
ADD_S	b,b,u7	11100bbb0uuuuuuu
ADD_S	SP,SP,u7	Synonymous with ADDL_S SP, SP, u9
ADD_S	R0,GP,s11	Synonymous with ADDL_S R0, GP, s11

Encodings supported by the CODE DENSITY option

ADD_S	R0,b,u6	01001bbb0UUU1uuu
ADD_S	R1,b,u6	01001bbb1UUU1uuu

ADDL

Function

Add long

Extension Group

Baseline

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

Syntax

ADDL<.f> dst, src1, src2

Description

The ADDL instruction computes the sum of its two 64-bit source operands src1 and src2, and places the 64-bit result in the destination dst. Any flag updates occur only if the set flags suffix (.f) is used.

Assembly Code Example

```
ADDL r1,r2,r3   ; Add contents of r2 with r3 and write result into r1
```

Pseudo Code

```
if cc==true then                        /* ADDL */
 dest = src1 + src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[6]
  C_flag = Carry()
  V_flag = Overflow()
```

Encodings

		Instruction Code
ADDL<.f>	a,b,c	01011bbb00000000FBBBCCCCCCAAAAAA

ADDL<.f>	a,b,u6	01011bbb01000000FBBBuuuuuuAAAAAA
ADDL<.f>	b,b,s12	01011bbb10000000FBBBssssssSSSSSS
ADDL<.cc><.f>	b,b,c	01011bbb11000000FBBBCCCCCC0QQQQQ
ADDL<.cc><.f>	b,b,u6	01011bbb11000000FBBBuuuuuu1QQQQQ
ADDL_S	b,b,c	01111bbbccc00001
ADDL_S	h,h,limm	01110001hhh110HH
ADDL_S	h,PCL,limm	01110011hhh110HH
ADDL_S	b,sp,u7	11000bbb100uuuuu
ADDL_S	SP,SP,u9	11000UU0101uuuuu
ADDL_S	R0,GP,s11	1100111sssssssss

ADDHL

Function

Add high long with 32-bit shift on second source

Extension Group

Baseline

Operation

if (cc) a = b + (c << 32)

Instruction Format

op a, b, c

STATUS32 Flags Affected

None

Syntax

ADDHL<.cc> a, b, c

Description

The least-significant 32 bits of the source operand (c) are shifted left 32 bit-positions before being added to the first source operand (b), to produce a 64-bit result.

Assembly Code Example

```
ADDHL r1,r1,0x87654321 ; r1 ← r1 + 0x8765432100000000 (limm opd)

ADDHL r0,r2,-4 ; r0 ← r2 + 0xfffffffc00000000 (s12 opd)
```

Pseudo Code

```
if (cc == true) {
dest[63:0] = src1[63:0] + (src2[31:0] << 32)
}
```

Instruction Encodings

		Instruction Code
ADDHL	a,b,c	01011bbb001011100BBBCCCCCCaaaaaa
ADDHL	a,b,u6	01011bbb011011100BBBuuuuuuaaaaaa
ADDHL	b,b,s12	01011bbb101011100BBBssssssSSSSSS
ADDHL<.cc>	b,b,c	01011bbb111011100BBBCCCCCC0QQQQQ
ADDHL<.cc>	b,b,u6	01011bbb111011100BBBuuuuuu1QQQQQ
ADDHL_S	h,h,LIMM	01110001hhh010HH
ADDHL_S	h,PCL,LIMM	01110011hhh010HH

ADD1

Function

Addition with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b + (c << 1);

Instruction Format

op a, b, c

Syntax Example

ADD1<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 1. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD1 */
if cc==true then
 shiftedsrc2 = src2 << 1
 dest = src1 + shiftedsrc2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31])
```

Assembly Code Example

```
ADD1 r1,r2,r3   ; Add contents of r3 shifted left one bit to r2 and
                ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
ADD1<.f>	a,b,c	00100bbb00010100FBBBCCCCCCAAAAAA
ADD1<.f>	a,b,u6	00100bbb01010100FBBBuuuuuuAAAAAA
ADD1<.f>	b,b,s12	00100bbb10010100FBBBssssssSSSSSS
ADD1<.cc><.f>	b,b,c	00100bbb11010100FBBBCCCCCC0QQQQQ
ADD1<.cc><.f>	b,b,u6	00100bbb11010100FBBBuuuuuu1QQQQQ
ADD1_S	b,b,c	01111bbbccc10100

ADD1L

Function

Long Addition with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b + (c << 1);

Instruction Format

op a, b, c

Syntax Example

ADD1L<.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 ADDL part of the instruction

Description

Add 64-bit source operand 1, b, to a scaled version of 64-bit source operand 2, c left shifted by 1. Place the result in the 64-bit destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD1L */
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[63]
  C_flag = Carry()
  V_flag = (src1[63] AND shiftedsrc2[63] and NOT dest[63] ) OR
( NOT src1[63] AND NOT shiftedsrc2[63] and dest[63])
```

Assembly Code Example

```
ADD1L r1,r2,r3   ; Add contents of r3 shifted left one bit to r2 and
                 ;write result into r1
```

Syntax and Encoding

		Instruction Code
ADD1L<.f>	a,b,c	01011bbb00010100FBBBCCCCCCAAAAAA
ADD1L<.f>	a,b,u6	01011bbb01010100FBBBuuuuuuAAAAAA
ADD1L<.f>	b,b,s12	01011bbb10010100FBBBssssssSSSSSS
ADD1L<.cc><.f>	b,b,c	01011bbb11010100FBBBCCCCCC0QQQQQ
ADD1L<.cc><.f>	b,b,u6	01011bbb11010100FBBBuuuuuu1QQQQQ

ADD2

Function

Addition with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b + (c << 2);

Instruction Format

op a, b, c

Syntax Example

ADD2<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 2. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD2 */
if cc==true then
 shiftedsrc2 = (src2 << 2)
 dest = src1 + shiftedsrc2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31])
```

Assembly Code Example

```
ADD2 r1,r2,r3      ; Add contents of r3 shifted left two bits to r2 and
                   ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
ADD2<.f>	a,b,c	00100bbb00010101FBBBCCCCCCAAAAAA
ADD2<.f>	a,b,u6	00100bbb01010101FBBBuuuuuuAAAAAA
ADD2<.f>	b,b,s12	00100bbb10010101FBBBssssssSSSSSS
ADD2<.cc><.f>	b,b,c	00100bbb11010101FBBBCCCCCC0QQQQQ
ADD2<.cc><.f>	b,b,u6	00100bbb11010101FBBBuuuuuu1QQQQQ
ADD2_S	b,b,c	01111bbbccc10101

ADD2L

Function

Long Addition with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b + (c << 2);

Instruction Format

op a, b, c

Syntax Example

ADD2L<.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 ADDL part of the instruction

Description

Add 64-bit source operand 1, b, to a scaled version of 64-bit source operand 2, c left shifted by 2. Place the 64-bit result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD2L */
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[63]
  C_flag = Carry()
  V_flag = (src1[63] AND shiftedsrc2[63] and NOT dest[63] ) OR
( NOT src1[63] AND NOT shiftedsrc2[63] and dest[63])
```

Assembly Code Example

```
ADD2L r1,r2,r3     ; Add contents of r3 shifted left two bits to r2 and
                   ;write result into r1
```

Syntax and Encoding

		Instruction Code
ADD2L<.f>	a,b,c	01011bbb00010101FBBBCCCCCCAAAAAA
ADD2L<.f>	a,b,u6	01011bbb01010101FBBBuuuuuuAAAAAA
ADD2L<.f>	b,b,s12	01011bbb10010101FBBBssssssSSSSSS
ADD2L<.cc><.f>	b,b,c	01011bbb11010101FBBBCCCCCC0QQQQQ

ADD3

Function

Addition with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b + (c << 3);

Instruction Format

op a, b, c

Syntax Example

ADD3<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the ADD part of the instruction

Description

Add source operand 1, b, to a scaled version of source operand 2, c left shifted by 3. Place the result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD3 */
if cc==true then
 shiftedsrc2 = src2 << 3
 dest = src1 + shiftedsrc2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = (src1[31] AND shiftedsrc2[31] and NOT dest[31] ) OR
( NOT src1[31] AND NOT shiftedsrc2[31] and dest[31])
```

Assembly Code Example

```
ADD3 r1,r2,r3   ; Add contents of r3 shifted left three bits to r2 and
                ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
ADD3<.f>	a,b,c	00100bbb00010110FBBBCCCCCCAAAAAA
ADD3<.f>	a,b,u6	00100bbb01010110FBBBuuuuuuAAAAAA
ADD3<.f>	b,b,s12	00100bbb10010110FBBBssssssSSSSSS
ADD3<.cc><.f>	b,b,c	00100bbb11010110FBBBCCCCCC0QQQQQ
ADD3<.cc><.f>	b,b,u6	00100bbb11010110FBBBuuuuuu1QQQQQ
ADD3_S	b,b,c	01111bbbccc10110

ADD3L

Function

Long Addition with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b + (c << 3);

Instruction Format

op a, b, c

Syntax Example

ADD3L<.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 64-bit source operand 1, b, to a scaled version of 64-bit source operand 2, c left shifted by 3. Place the result in the 64-bit destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
/* ADD3L */
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[63]
  C_flag = Carry()
  V_flag = (src1[63] AND shiftedsrc2[63] and NOT dest[63] ) OR
( NOT src1[31] AND NOT shiftedsrc2[63] and dest[63])
```

Assembly Code Example

```
ADD3L r1,r2,r3   ; Add contents of r3 shifted left three bits to r2 and
                 ; write result into r1
```

Syntax and Encoding

		Instruction Code
ADD3L<.f>	a,b,c	01011bbb00010110FBBBCCCCCCAAAAAA
ADD3L<.f>	a,b,u6	01011bbb01010110FBBBuuuuuuAAAAAA
ADD3L<.f>	b,b,s12	01011bbb10010110FBBBssssssSSSSSS
ADD3L<.cc><.f>	b,b,c	01011bbb11010110FBBBCCCCCC0QQQQQ
ADD3L<.cc><.f>	b,b,u6	01011bbb11010110FBBBuuuuuu1QQQQQ

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 Table 8-7 and Table 8-8.

		Instruction Code
AEX	b,[c]	00100bbb00100111RBBBCCCCCCRRRRRR
AEX	b,[u6]	00100bbb01100111RBBBuuuuuuRRRRRR
AEX	b,[s12]	00100bbb10100111RBBBssssssSSSSSS
AEX<.cc>	b,[c]	00100bbb11100111RBBBCCCCCC0QQQQQ
AEX<.cc>	b,[u6]	00100bbb11100111RBBBuuuuuu1QQQQQ

AEXL

Function

Swap contents of an auxiliary register with a core register.

Extension Group

Baseline

Instruction Format

op b, [c]

Syntax Example

AEXL<.cc> b, [c]

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Swaps contents of an auxiliary register with a 64-bit core register. This instruction is used by the micro-operations sequencer within the exception/interrupt entry/exit sequences. The b register operand of AEXL 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 AEXL instruction behaves as the union of the LRL and SRL instructions, with the exception that an AEXL can be conditional whereas LRL/SRL are not.

Pseudo Code

```
if cc==true then                    /* AEXL */
   tmp = AuxRead64 (src2)
   AuxWrite64 (src2, b)
   b = tmp
```

Assembly Code Example

```
AEXL R28, [AUX_User_SP]    ; swap User and kernel stack pointers
```

Syntax and Encoding

		Instruction Code
AEXL	b,[c]	01011bbb00100111 0BBBCCCCCCRRRRRR
AEXL	b,[u6]	01011bbb01100111 0BBBuuuuuuRRRRRR
AEXL	b,[s12]	01011bbb10100111 0BBBssssssSSSSSS
AEXL<.cc>	b,[c]	01011bbb11100111 0BBBCCCCCC0QQQQQ
AEXL<.cc>	b,[u6]	01011bbb11100111 0BBBuuuuuu1QQQQQ

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, seeTable 8-7 and Table 8-8.

		Instruction Code
AND<.f>	a,b,c	00100bbb00000100FBBBCCCCCCAAAAAA
AND<.f>	a,b,u6	00100bbb01000100FBBBuuuuuuAAAAAA
AND<.f>	b,b,s12	00100bbb10000100FBBBssssssSSSSSS
AND<.cc><.f>	b,b,c	00100bbb11000100FBBBCCCCCC0QQQQQ
AND<.cc><.f>	b,b,u6	00100bbb11000100FBBBuuuuuu1QQQQQ
AND_S	b,b,c	01111bbbccc00100

ANDL

Function

Bitwise AND Long Operation

Extension Group

Baseline

Operation

if (cc) a = b & c;

Instruction Format

op a, b, c

Syntax Example

ANDL<.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 64-bit source operand 1 (b) with 64-bit source operand 2 (c) with the result written to the 64-bit destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                        /* ANDL */
 dest = src1 AND src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
ANDL r1,r2,r3    ; AND contents of r2 with r3 and write result into r1
```

Syntax and Encoding

		Instruction Code
ANDL<.f>	a,b,c	01011bbb00000100FBBBCCCCCCAAAAAA
ANDL<.f>	a,b,u6	01011bbb01000100FBBBuuuuuuAAAAAA
ANDL<.f>	b,b,s12	01011bbb10000100FBBBssssssSSSSSS
ANDL<.cc><.f>	b,b,c	01011bbb11000100FBBBCCCCCC0QQQQQ
ANDL<.cc><.f>	b,b,u6	01011bbb11000100FBBBuuuuuu1QQQQQ
ANDL_S	b,b,c	01111bbbccc01000

ASL

Function

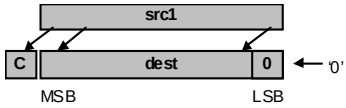
Arithmetic Shift Left

Extension Group

Baseline

Operation

b = (signed) c << 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 Table 8-7 and Table 8-8.

		Instruction Code
ASL<.f>	b,c	00100bbb00101111FBBBCCCCCC000000
ASL<.f>	b,u6	00100bbb01101111FBBBuuuuuu000000
ASL_S	b,c	01111bbbccc11011

ASLL

Function

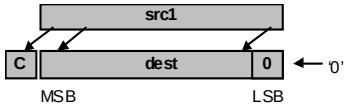
Arithmetic Shift Left Long

Extension Group

Baseline

Operation

b = (signed) c << 1;



Instruction Format

op b, c

Syntax Example

ASLL<.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 64-bit source operand (c) by one and place the result into the 64-bit destination register (b). An ASLL 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                    /* ASLL */
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[63]
 C_flag = Carry()
 V_flag = Overflow()
```

Assembly Code Example

```
ASLL r1,r2   ; Arithmetic shift left contents of r2 by one bit and
             ; write result into r1
```

Syntax and Encoding

		Instruction Code
ASLL<.f>	a,b,c	01011bbb00100000FBBBCCCCCCAAAAAA
ASLL<.f>	a,b,u6	01011bbb01100000FBBBuuuuuuAAAAAA
ASLL<.f>	b,b,s12	01011bbb10100000FBBBssssssSSSSSS
ASLL<.f>	b,b,c	01011bbb11100000FBBBCCCCCC0QQQQQ
ASLL<.f>	b,b,u6	01011bbb11100000FBBBuuuuuu1QQQQQ
ASLL<.f>	b,c	01011bbb00101111FBBBCCCCCC000000
ASLL<.f>	b,u6	01011bbb01101111FBBBuuuuuu000000

ASL Multiple

Function

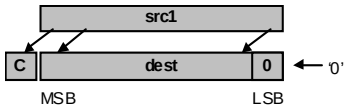
Multiple Arithmetic Shift Left

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
ASL<.f>	a,b,c	00101bbb00000000FBBBCCCCCCAAAAAA
ASL<.f>	a,b,u6	00101bbb01000000FBBBuuuuuuAAAAAA
ASL<.f>	b,b,s12	00101bbb10000000FBBBssssssSSSSSS
ASL<.cc><.f>	b,b,c	00101bbb11000000FBBBCCCCCC0QQQQQ
ASL<.cc><.f>	b,b,u6	00101bbb11000000FBBBuuuuuu1QQQQQ
ASL_S	b,b,c	01111bbbccc11000
ASL_S	b,b,u5	10111bbb000uuuuu

ASLL Multiple

Function

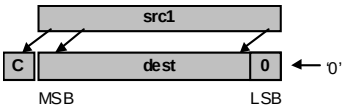
Multiple Arithmetic Shift Left Long

Extension Group

Baseline

Operation

if (cc) a = (signed) b << c;



Instruction Format

op a, b, c

Syntax Example

ASLL<.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 the 64-bit source operand b by c places and place the result in the 64-bit destination register. Only the bottom 6 bits of c are used as the shift value. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                        /* ASLL */
 dest = src1 << (src2 & 63)                             /* Multiple */
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = if src2==0 then 0 else src1[64-src2]
```

Assembly Code Example

```
ASLL r1,r2,r3    ; Arithmetic shift left contents of r2 by r3 bits
                 ; and write result into r1
```

Syntax and Encoding

		Instruction Code
ASLL<.f>	a,b,c	01011bbb00100000FBBBCCCCCCAAAAAA
ASLL<.f>	a,b,u6	01011bbb01100000FBBBuuuuuuAAAAAA
ASLL<.f>	b,b,s12	01011bbb10100000FBBBssssssSSSSSS
ASLL<.cc><.f>	b,b,c	01011bbb11100000FBBBCCCCCC0QQQQQ
ASLL<.cc><.f>	b,b,u6	01011bbb11100000FBBBuuuuuu1QQQQQ

ASR

Function

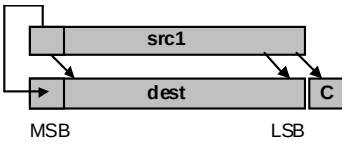
Arithmetic Shift Right

Extension Group

Baseline

Operation

b = (signed) c >> 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, seeTable 8-7 and Table 8-8.

		Instruction Code
ASR<.f>	b,c	00100bbb00101111FBBBCCCCCC000001
ASR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000001
ASR_S	b,c	01111bbbccc11100

ASRL

Function

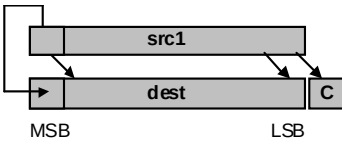
Arithmetic Shift Right Long

Extension Group

Baseline

Operation

b = (signed) c >> 1;



Instruction Format

op b, c

Syntax Example

ASRL<.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 64-bit source operand (c) by one and place the result into the 64-bit 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                      /* ASRL */
if src[63]==1 then dest[63] = 1
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[63]
 C_flag = src[0]
```

Assembly Code Example

```
ASRL r1,r2    ; Arithmetic shift right contents of r2 by one bit and
              ; write result into r1
```

Syntax and Encoding

		Instruction Code
ASRL<.f>	a,b,c	01011bbb00100010FBBBCCCCCCAAAAAA
ASRL<.f>	a,b,u6	01011bbb01100010FBBBuuuuuuAAAAAA
ASRL<.f>	b,b,s12	01011bbb10100010FBBBssssssSSSSSS
ASRL<.f>	b,b,c	01011bbb11100010FBBBCCCCCC0QQQQQ
ASRL<.f>	b,b,u6	01011bbb11100010FBBBuuuuuu1QQQQQ
ASRL<.f>	b,c	01011bbb00101111FBBBCCCCCC000001
ASRL<.f>	b,u6	01011bbb01101111FBBBuuuuuu000001

ASR multiple

Function

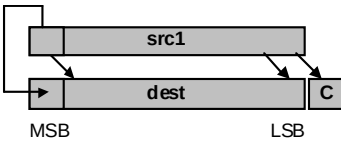
Multiple Arithmetic Shift Right

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
ASR<.f>	a,b,c	00101bbb00000010FBBBCCCCCCAAAAAA
ASR<.f>	a,b,u6	00101bbb01000010FBBBuuuuuuAAAAAA
ASR<.f>	b,b,s12	00101bbb10000010FBBBssssssSSSSSS
ASR<.cc><.f>	b,b,c	00101bbb11000010FBBBCCCCCC0QQQQQ
ASR<.cc><.f>	b,b,u6	00101bbb11000010FBBBuuuuuu1QQQQQ
ASR_S	b,b,c	01111bbbccc11010
ASR_S	b,b,u5	10111bbb010uuuuu

ASRL Multiple

Function

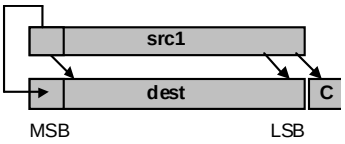
Multiple Arithmetic Shift Right Long

Extension Group

Baseline

Operation

if (cc) a = (signed) b >> c;



Instruction Format

op a, b, c

Syntax Example

ASRL<.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 64-bit operand b right by c places and place the result in the 64-bit destination register a. Only the bottom 6 bits of c are used as the shift value. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                          /* ASRL */
 dest = ((signed)src1) >> (src2 & 63)                     /* Multiple */
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = if src2==0 then 0 else src1[src2-1]
```

Assembly Code Example

```
ASRL r1,r2,r3   ; Arithmetic shift right contents of r2 by r3 bits
                ; and write result into r1
```

Syntax and Encoding

		Instruction Code
ASRL<.f>	a,b,c	01011bbb00100010FBBBCCCCCCAAAAAA
ASRL<.f>	a,b,u6	01011bbb01100010FBBBuuuuuuAAAAAA
ASRL<.f>	b,b,s12	01011bbb10100010FBBBssssssSSSSSS
ASRL<.cc><.f>	b,b,c	01011bbb11100010FBBBCCCCCC0QQQQQ
ASRL<.cc><.f>	b,b,u6	01011bbb11100010FBBBuuuuuu1QQQQQ

ASR16

Function

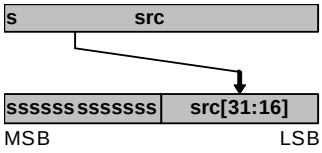
Arithmetic Shift Right 16

Extension Group

Baseline

Operation

b = c >> 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 Table 8-7 and Table 8-8.

		Instruction Code
ASR16<.f>	b,c	00101bbb00101111FBBBCCCCCC001100
ASR16<.f>	b,u6	00101bbb01101111FBBBuuuuuu001100

ASR8

Function

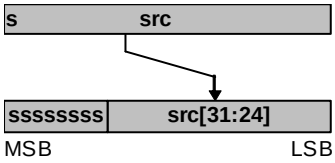
Arithmetic Shift Right 8

Extension Group

Baseline

Operation

b = c >> 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 Table 8-7 and Table 8-8.

		Instruction Code
ASR8<.f>	b,c	00101bbb00101111FBBBCCCCCC001101
ASR8<.f>	b,u6	00101bbb01101111FBBBuuuuuu001101

ATLD

Function

Atomic fetch and operate instruction

Extension Group

Baseline

Operation

See the pseudo code section

Instruction Format

op b,[c]

Syntax Example

ATLD<.op><.aq.rl> b,c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Each of the AMOs provides an option to specify acquire semantics on the memory read and release semantics on the memory write. This is indicated by the optional suffix `.aq.rl` applied to the instruction mnemonic.

The ARCv3 ISA continues to support the uncached version of the EX instruction (EX.DI), but within ARCv3 the ARC64 ISA does not provide uncached versions of the new AMOs nor an uncached version of EXL.

All AMOs perform an atomic sequence in which the memory location defined by Rc is read; the value read is then combined with source register Rb using the operator of that instruction; the result is written back to memory at address Rc; and the original value from memory is returned to destination register Rb. The EX instruction is a degenerate form of AMO in which the value written to memory is simply the Rb source operand. This has the effect of swapping the contents of Rb and the contents of memory at the location given by Rc.

The memory address of an AMO must be aligned to the size of the data object it references. This is primarily to ensure that the object referenced by an atomic memory operation never straddles a cache-line boundary. If an AMO has a non-aligned address it always raises an EV_Misaligned exception.

Pseudo Code

Instruction		Semantics
ex<.aq.rl>	Rb,[Rc]	atomic (tmp ←mem[Rc]; mem[Rc] ← Rb; Rb ← tmp)
atld.add<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = add(Rb, mem[Rc]); Rb ← tmp)
atld.or<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = or(Rb, mem[Rc]); Rb ← tmp)
atld.and<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = and(Rb, mem[Rc]); Rb ← tmp)
atld.xor<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = xor(Rb, mem[Rc]); Rb ← tmp)
atld.minu<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = umin(Rb, mem[Rc]) ; Rb ← tmp)
atld.maxu<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = umax(Rb, mem[Rc]) ; Rb ← tmp)
atld.min<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = min(Rb, mem[Rc]) ; Rb ← tmp)
atld.max<.aq.rl>	Rb,[Rc]	atomic (tmp ← mem[Rc]; mem[Rc] = max(Rb, mem[Rc]) ; Rb ← tmp)

Assembly Code Example

```
ATLD.op.aq.rl r1,r2          ;  perform atomic memory operations on values in the
                             memory.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Table 11-2 Encodings for 32-bit AMO Instructions

ex	b,[c]	00100bbb00101111 0BBBCCCCCC001100
ex	b,[u6]	00100bbb01101111 0BBBuuuuuu001100
ex.di	b,[c]	00100bbb00101111 1BBBCCCCCC001100
ex.di	b,[u6]	00100bbb01101111 1BBBuuuuuu001100
ex.aq.rl	b,[c]	00100bbb00101111 0BBBCCCCCC001110
ex.aq.rl	b,[u6]	00100bbb01101111 0BBBuuuuuu001110
atld<.op><.aq.rl>	b,[c]	00100bbb00101111 mBBBCCCCCC110iii

Table 11-3 ATLD Sub-Opcode Encodings

I[2:0]	Atomic Load Instruction
000	atld.add
001	atld.or
010	atld.and
011	atld.xor
100	atld.minu

Table 11-3 ATLD Sub-Opcode Encodings

I[2:0]	Atomic Load Instruction
101	atld.maxu
110	atld.min
111	atld.max

ATLDL

Function

Atomic fetch and operate instruction long

Extension Group

Baseline

Operation

See the pseudo code section

Instruction Format

op b,[c]

Syntax Example

ATLDL<.op><.aq.rl> b,c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

This instruction is an atomic memory operations (AMO) instruction that operates on 64-bit values in the memory. Each of the AMOs provides an option to specify acquire semantics on the memory read and release semantics on the memory write. These operations are indicated by the optional suffix .aq.rl applied to the instruction mnemonic.

All AMOs perform an atomic sequence in which the memory location defined by Rc is read; the value read is then combined with source register Rb using the operator of that instruction; the result is written back to memory at address Rc; and the original value from memory is returned to destination register Rb. The EX instruction is a degenerate form of AMO in which the value written to memory is simply the Rb source operand. This has the effect of swapping the contents of Rb and the contents of memory at the location given by Rc.

The memory address of an AMO must be aligned to the size of the data object it references. This is primarily to ensure that the object referenced by an atomic memory operation never straddles a cache-line boundary. If an AMO has a non-aligned address it always raises an EV_Misaligned exception.

Pseudo Code

```
ATLDL.add<.aq.rl>      b,[c]    atomic (tmp <- mem[Rc]; mem[Rc] += Rb; Rb <- tmp)
ATLDL.or<.aq.rl>       b,[c]    atomic (tmp <- mem[Rc]; mem[Rc] |= Rb; Rb <- tmp)
ATLDL.and<.aq.rl>      b,[c]    atomic (tmp <- mem[Rc]; mem[Rc] &= Rb; Rb <- tmp)
ATLDL.xor<.aq.rl>      b,[c]    atomic (tmp <- mem[Rc]; mem[Rc] ^= Rb; Rb <- tmp)
ATLDL.minu<.aq.rl>     b,[c]    atomic (tmp <- mem[Rc]; mem[Rc] = umin(Rb, mem[Rc]) ; Rb <- tmp)
ATLDL.maxu<.aq.rl>     b,[c]    atomic (tmp <- mem[Rc]; mem[Rc] = umax(Rb, mem[Rc]) ; Rb <- tmp)
ATLDL.min<.aq.rl>      b,[c]    atomic (tmp <- mem[Rc]; mem[Rc] = min(Rb, mem[Rc]) ; Rb <- tmp)
ATLDL.max<.aq.rl>      b,[c]    atomic (tmp <- mem[Rc];  mem[Rc] = max(Rb, mem[Rc]) ; Rb <- tmp)
```

Assembly Code Example

```
ATLDL.op.aq.rl r1,r2        ;  perform atomic memory operations on values in the
                            memory.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
ATLDL.add	b,[c]	01011bbb00101111 0BBBCCCCCC110000
ATLDL.or	b,[c]	01011bbb00101111 0BBBCCCCCC110001
ATLDL.and	b,[c]	01011bbb00101111 0BBBCCCCCC110010
ATLDL.xor	b,[c]	01011bbb00101111 0BBBCCCCCC110011
ATLDL.minu	b,[c]	01011bbb00101111 0BBBCCCCCC110110
ATLDL.maxu	b,[c]	01011bbb00101111 0BBBCCCCCC110101
ATLDL.min	b,[c]	01011bbb00101111 0BBBCCCCCC110110
ATLDL.max	b,[c]	01011bbb00101111 0BBBCCCCCC110111
ATLDL.add<.aq.rl>	b,[c]	01011bbb00101111 1BBBCCCCCC110000
ATLDL.or<.aq.rl>	b,[c]	01011bbb00101111 1BBBCCCCCC110001
ATLDL.and<.aq.rl>	b,[c]	01011bbb00101111 1BBBCCCCCC110010
ATLDL.xor<.aq.rl>	b,[c]	01011bbb00101111 1BBBCCCCCC110011

```
ATLDL.minu<.aq.rl>     b,[c]    01011bbb001011111BBBCCCCCC110110
ATLDL.maxu<.aq.rl>     b,[c]    01011bbb001011111BBBCCCCCC110101
ATLDL.min<.aq.rl>      b,[c]    01011bbb001011111BBBCCCCCC110110
ATLDL.max<.aq.rl>      b,[c]    01011bbb001011111BBBCCCCCC110111
```

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 8-13 on page 238 describes the delay slot modes, .d.

Table 8-9 on page 233 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.

The processor raises an Illegal Instruction Sequence exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- 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 Table 8-7 and Table 8-8.

		Instruction Code
B<.d>	s25	00000ssssssssss1SSSSSSSSSSNRtttt

B_S

Function

16-Bit Branch

Extension Group

Baseline

Operation

PC = (PCL+s10)

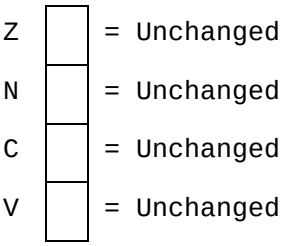
Instruction Format

op s10

Syntax Example

B_S s10

STATUS32 Flags Affected



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.

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 Table 8-7 and Table 8-8.

		Instruction Code
B_S	s10	1111000ssssssssss

BBIT0

Function

Branch on Bit Test Clear

Extension Group

Baseline

Operation

if ((b AND (1<<c)) == 0) PC = PCL+s9;

Instruction Format

op b, c, s9

Syntax Example

BBIT0<.d><.T> b, c, s9

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Test a bit within source operand 1 (b) to see if the bit is clear (0). Source operand 2 (c) explicitly specifies the position of the bit that is to be tested within source operand 1 (b). Only the bottom 5 bits of operand 2 are used as the bit position. If the bit is clear, the branch is taken. The branch target is computed as the sum of the current 32-bit word-aligned PC and the signed half-word displacement value specified by the 9-bit literal source operand (s9).

Because the delay slot mode controls the execution of the instruction that is in the delay slot, the instruction must never be the target of any branch or jump instruction. The status flags are not updated by this instruction.

Table 8-13 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.

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 Table 8-7 and Table 8-8.

		Instruction Code
BBIT0<.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN00110
BBIT0<.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN10110

BBIT0L

Function

Branch on Bit Test Clear

Extension Group

Baseline

Operation

if ((b AND (1<<c)) == 0) PC = PCL+s9;

Instruction Format

op b, c, s9

Syntax Example

BBIT0l<.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 six 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 8-13 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.

Pseudo Code

```
if (src1 & (1 << (src2 & 63)))==0    /* BBIT0l */
then
 if N=1 then
  DelaySlot(nPC)
 KillDelaySlot(nPC + 1)
 PC = PCL + s9
else
 PC = nPC
```

Assembly Code Example

```
BBIT0L r1,9,label    ; Branch to label if bit 9
                     ; of r1 is clear
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BBIT0L<.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN01110
BBIT0L<.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN11110

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).

In ARC64 only the bottom 6 bits of operand 2 are used.

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 8-13 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.

Pseudo Code

```
if N=1 then                                                  /* BBIT1 */
  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 Table 8-7 and Table 8-8.

		Instruction Code
BBIT1<.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN00111
BBIT1<.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN10111

BBIT1L

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).

The bottom 6 bits of operand 2 are used.

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 8-13 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.

Pseudo Code

```
M = if STATUS32.M == 1 then 63 else 31                  /* BBIT1L */
if (src1 & (1 << (src2 & M)))!=0 then
 if N=1 then
  DelaySlot(nPC)
 KillDelaySlot(dPC)
 PC = cPCL + rd
else
 PC = nPC
```

Assembly Code Example

```
BBIT1L r1,9,label   ; Branch to label if bit 9  of r1 is set
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BBIT1L<.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN01111
BBIT1L<.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN11111

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 8-13.

The condition codes, cc, are described in Table 8-9.

The Bcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction.

The processor raises an Illegal Instruction Sequence exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- 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 Table 8-7 and Table 8-8.

		Instruction Code
B<cc><.d>	s21	00000ssssssssss0SSSSSSSSSSNQQQQQ

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 8-9 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.

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 Table 8-7 and Table 8-8.

		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 & ~(1<< c));

Instruction Format

op a, b, c

Syntax Example

BCLR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Clear (0) an individual bit within the value that is specified by the source operand 1 (b). The source operand 2 (c) contains a value that explicitly defines the bit-position that is to be cleared in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The result is written into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                          /* BCLR */
 dest = src1 AND NOT(1 << (src2 & 31))
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
BCLR r1,r2,r3    ; Clear bit r3 of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BCLR<.f>	a,b,c	00100bbb00010000FBBBCCCCCCAAAAAA
BCLR<.f>	a,b,u6	00100bbb01010000FBBBuuuuuuAAAAAA
BCLR<.f>	b,b,s12	00100bbb10010000FBBBssssssSSSSSS
BCLR<.cc><.f>	b,b,c	00100bbb11010000FBBBCCCCCC0QQQQQ
BCLR<.cc><.f>	b,b,u6	00100bbb11010000FBBBuuuuuu1QQQQQ
BCLR_S	b,b,u5	10111bbb101uuuuu

BCLRL

Function

Bit Clear Long

Extension Group

Baseline

Operation

if (cc) then a = (b & ~(1<< c));

Instruction Format

op a, b, c

Syntax Example

BCLRL<.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 64-bit value specified by source operand 1 (b). Source operand 2 (c) contains a value that explicitly defines the position of the bit that is to be cleared in source operand 1 (b). Only the bottom 6 bits of c are used as the bit value. The result is written into the 64-bit destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                          /* BCLRL */
 dest = src1 AND NOT(1 << (src2 & 63))
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
BCLRL r1,r2,r3   ; Clear bit r3 of r2 and write result into r1
```

Syntax and Encoding

		Instruction Code
BCLRL<.f>	a,b,c	01011bbb00010000FBBBCCCCCCAAAAAA
BCLRL<.f>	a,b,u6	01011bbb01010000FBBBuuuuuuAAAAAA
BCLRL<.f>	b,b,s12	01011bbb10010000FBBBssssssSSSSSS
BCLRL<.cc><.f>	b,b,c	01011bbb11010000FBBBCCCCCC0QQQQQ
BCLRL<.cc><.f>	b,b,u6	01011bbb11010000FBBBuuuuuu1QQQQQ

BI

Function

Branch Indexed 32-bit Full-Word Table

Extension Group

Baseline

Operation

PC = next_PC + (c << 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 Table 8-7 and Table 8-8.

		Instruction Code
BI	[c]	00100RRR00100100RRRCCCCCCRRRRRR

BIC

Function

Bitwise AND Operation with Inverted Source

Extension Group

Baseline

Operation

if (cc) a = b & ~c;

Instruction Format

op a, b, c

Syntax Example

BIC<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise AND of source operand 1 (b) with the inverse of source operand 2 (c) with the result written to the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                          /* BIC */
 dest = src1 AND NOT src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
BIC r1,r2,r3    ; AND r2 with the NOT of r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BIC<.f>	a,b,c	00100bbb00000110FBBBCCCCCCAAAAAA
BIC<.f>	a,b,u6	00100bbb01000110FBBBuuuuuuAAAAAA
BIC<.f>	b,b,s12	00100bbb10000110FBBBssssssSSSSSS
BIC<.cc><.f>	b,b,c	00100bbb11000110FBBBCCCCCC0QQQQQ
BIC<.cc><.f>	b,b,u6	00100bbb11000110FBBBuuuuuu1QQQQQ
BIC_S	b,b,c	01111bbbccc00110

BICL

Function

Bitwise AND Long Operation with Inverted Source

Extension Group

Baseline

Operation

if (cc) a = b & ~c;

Instruction Format

op a, b, c

Syntax Example

BIC<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise AND of 64-bit source operand 1 (b) with the inverse of 64-bit source operand 2 (c), with the result written to the 64-bit destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                          /* BICL */
 dest = src1 AND NOT src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
BICL r1,r2,r3    ; AND r2 with the NOT of r3 and write result into r1
```

Syntax and Encoding

		Instruction Code
BICL<.f>	a,b,c	01011bbb00000110FBBBCCCCCCAAAAAA
BICL<.f>	a,b,u6	01011bbb01000110FBBBuuuuuuAAAAAA
BICL<.f>	b,b,s12	01011bbb10000110FBBBssssssSSSSSS
BICL<.cc><.f>	b,b,c	01011bbb11000110FBBBCCCCCC0QQQQQ
BICL<.cc><.f>	b,b,u6	01011bbb11000110FBBBuuuuuu1QQQQQ

BIH

Function

Branch Indexed to Half-Word Table

Extension Group

Baseline

Operation

PC = next_PC + (c << 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 Table 8-7 and Table 8-8.

Instruction Code

```
BIH   [c]   00100RRR00100101 0RRRCCCCCCRRRRRR
```

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 8-13.

The condition codes, cc, are described in Table 8-9.

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.

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
- Return from interrupt (RTIE)
- Any instruction with long-immediate data as a source operand

An Illegal Instruction Sequence exception is raised on a branch instruction if that branch instruction is specified with a delay-slot instruction and if the address of the instruction after that delay-slot instruction is equal to LP_END when STATUS32.L is 0.

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 Table 8-7 and Table 8-8.

```
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	00001sssssssss00SSSSSSSSSSNQQQQQ
BL<.d>	s25	00001sssssssss10SSSSSSSSSSNRtttt
BL_S	s13	11111ssssssssss

BMSK

Function

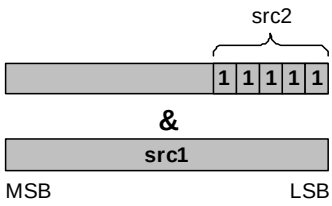
Bit Mask

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 Table 8-7 and Table 8-8.

		Instruction Code
BMSK<.f>	a,b,c	00100bbb00010011FBBBCCCCCCAAAAAA
BMSK<.f>	a,b,u6	00100bbb01010011FBBBuuuuuuAAAAAA
BMSK<.f>	b,b,s12	00100bbb10010011FBBBssssssSSSSSS
BMSK<.cc><.f>	b,b,c	00100bbb11010011FBBBCCCCCC0QQQQQ
BMSK<.cc><.f>	b,b,u6	00100bbb11010011FBBBuuuuuu1QQQQQ
BMSK_S	b,b,u5	10111bbb110uuuuu

BMSKL

Function

Bit Mask Long

Extension Group

Baseline

Operation

if (cc) then a = (b & ((1 << (c+1))-1);

Instruction Format

op a, b, c

Syntax Example

BMSKL<.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 64-bit mask value in which all bits up to and including the bit specified by operand 2 (c) are set to 1, and all other bits are 0. Only the bottom 6 bits of src2 are used to specify the bit mask.

A logical AND is performed with the mask value and 64-bit source operand (b). The result is written into the 64-bit destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                              /* BMSKL */
 dest = src1 AND ((1 << ((src2 & 63)+1))-1)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
BMSKL r1,r2,63    ; Do not mask any bits of r2, write result into r1
BMSKL r1,r2,7     ; Mask out the top 56 bits of r2 and write result into r1
BMSKL r1,r2,0     ; Mask out all except bit 0 of r2 and write result into r1
```

Syntax and Encoding

		Instruction Code
BMSKL<.f>	a,b,c	01011bbb00010011FBBBCCCCCCAAAAAA
BMSKL<.f>	a,b,u6	01011bbb01010011FBBBuuuuuuAAAAAA
BMSKL<.f>	b,b,s12	01011bbb10010011FBBBssssssSSSSSS
BMSKL<.cc><.f>	b,b,c	01011bbb11010011FBBBCCCCCC0QQQQQ
BMSKL<.cc><.f>	b,b,u6	01011bbb11010011FBBBuuuuuu1QQQQQ

BMSKN

Function

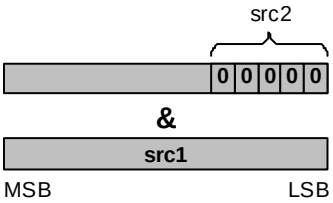
Bit Mask Negated

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 & 0xFFFFFE00)
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BMSKN<.f>	a,b,c	00100bbb00101100FBBBCCCCCCAAAAAA
BMSKN<.f>	a,b,u6	00100bbb01101100FBBBuuuuuuAAAAAA
BMSKN<.f>	b,b,s12	00100bbb10101100FBBBssssssSSSSSS
BMSKN<.cc><.f>	b,b,c	00100bbb11101100FBBBCCCCCC0QQQQQ
BMSKN<.cc><.f>	b,b,u6	00100bbb11101100FBBBuuuuuu1QQQQQ

BMSKNL

Function

Bit Mask Negated Long

Extension Group

Baseline

Operation

if (cc) then a = (b & (~0 << (c+1)));

Instruction Format

op a, b, c

Syntax Example

BMSKNL <.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 64-bit mask value in which all bits up to and including the bit specified by operand 2 (c) are set to 0, and all other bits are 1. Only the bottom 6 bits of src2 are used to specify the bit mask.

A logical AND is performed with the mask value and 64-bit source operand (b). The result is written into the 64-bit destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                    /*BMSKNL */
 dest = src1 AND ~((1 << ((src2 & 63)+1))-1)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
BMSKNL r1,r2,8    ; r1 is set to (r2 & 0xFFFFFFFF_FFFFFE00)
```

Syntax and Encoding

		Instruction Code
BMSKNL<.f>	a,b,c	01011bbb00101100FBBBCCCCCCAAAAAA
BMSKNL<.f>	a,b,u6	01011bbb01101100FBBBuuuuuuAAAAAA
BMSKNL<.f>	b,b,s12	01011bbb10101100FBBBssssssSSSSSS
BMSKNL<.cc><.f>	b,b,c	01011bbb11101100FBBBCCCCCC0QQQQQ
BMSKNL<.cc><.f>	b,b,u6	01011bbb11101100FBBBuuuuuu1QQQQQ

BRcc

Function

Compare and Branch

Extension Group

Baseline

Operation

if (cc(b,c)) PC = PCL+s9

Instruction Format

op b, c, s9

Syntax Example

BREQ<.d> b, c, s9

STATUS32 Flags Affected

None

Description

The BRcc instructions each perform one of six possible 32-bit relational tests on their operands, and if the result is true, the branch is taken. If a delay-slot is specified, using delay slot mode shown in the following table, 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 word-aligned PC plus the 16-bit half-word aligned displacement value given by the signed 9-bit literal operand (s9).

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)
```

Because the delay slot mode controls the execution of the instruction that is in the delay slot, the delay slot itself must never be the target of any branch or jump instruction.

Caution The BRcc instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, BRcc.D, BRccL.d, BBIT0.D or BBIT1.D instruction.

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BREQ <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN00000
BREQ <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN10000
BRNE <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN00001
BRNE <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN10001
BRLT <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN00010
BRLT <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN10010
BRGE <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN00011
BRGE <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN10011
BRLO <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN00100
BRLO <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN10100
BRHS <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN00101
BRHS <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN10101

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 (if u6 < 63)
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 (if u6 < 63)
```

```
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 (if u6 < 63)
BRLS<.d> b,c,s9       Encode as BRHS<.d> c,b,s9
BRLS<.d> b,u6,s9      Encode as BRLO b,u6+1,s9 (if u6 < 63)
```

BRccL

Function

Compare and Branch

Extension Group

Baseline

Operation

if (cc(b,c)) PC = PCL+s9

Instruction Format

op b, c, s9

Syntax Example

BREQL<.d> b, c, s9

STATUS32 Flags Affected

None

Description

The BRccL instructions each perform one of six possible 32-bit relational tests on their operands, and if the result is true, the branch is taken. If a delay-slot is specified, using delay slot mode shown in the following table, 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 word-aligned PC plus the 16-bit half-word aligned displacement value given by the signed 9-bit literal operand (s9).

The available branch conditions are shown in the following table:

Instruction	Description	Branch Condition
BREQL	Branch if Equal	if (b==c) PC ← (PCL +s9)
BRNEL	Branch if Not Equal	if (b!=c) PC ← (PCL + s9)
BRLTL	Branch if Less Than (Signed)	if (b<c) PC ← (PCL + s9)
BRGEL	Branch if Greater Than or Equal (Signed)	if (b>=c) PC ← (PCL + s9)
BRLOL	Branch if Lower Than (Unsigned)	if (b<c) PC ← (PCL + s9)
BRHSL	Branch if Higher Than or Same (Unsigned)	if (b>=c) PC ← (PCL + s9)

Because the delay slot mode controls the execution of the instruction that is in the delay slot, the delay slot itself must never be the target of any branch or jump instruction.

Caution The BRcc instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, BRcc.D, BRccL.d, BBIT0.D or BBIT1.D instruction.

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BREQL <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN01000
BREQL <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN11000
BRNEL <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN01001
BRNEL <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN11001
BRLTL <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN01010
BRLTL <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN11010
BRGEL <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN01011
BRGEL <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN11011
BRLOL <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN01100
BRLOL <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN11100
BRHSL <.d>	b,c,s9	00001bbbssssssss1SBBBCCCCCCN01101
BRHSL <.d>	b,u6,s9	00001bbbssssssss1SBBBuuuuuuN11101
BRNEL_S	b,0,s8	11101bbb0sssssss
BREQL_S	b,0,s8	11101bbb1sssssss

Additional Pseudo Syntax and Encoding

The following pseudo-instructions for missing conditions are available using existing encodings:

```
BRGTL<.d>    b,c,s9      Encode as BRLTL<.d> c,b,s9
BRGTL<.d>    b,u6,s9     Encode as BRGEL<.d> b,u6+1,s9 (if u6 < 63)
BRHIL<.d>    b,c,s9      Encode as BRLOL<.d> c,b,s9
BRHIL<.d>    b,u6,s9     Encode as BRHSL<.d> b,u6+1,s9 (if u6 < 63)
BRLEL<.d>    b,c,s9      Encode as BRGEL<.d> c,b,s9
```

```
BRLEL<.d>    b,u6,s9      Encode as BRLTL<.d> b,u6+1,s9 (if u6 < 63)
BRLSL<.d>    b,c,s9       Encode as BRHSL<.d> c,b,s9
BRLSL<.d>    b,u6,s9      Encode as BRLOL b,u6+1,s9 (if u6 < 63)
```

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.

Note	The breakpoint instruction sets the BH bit in the Debug register, which allows the debugger to determine what caused the ARCv3-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 ARCv3-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 ARCv3 provides both a 32-bit (BRK) and a 16-bit (BRK_S) encoding of the breakpoint instruction.

The breakpoint instruction can be placed anywhere in a program, including the delay slot of branch or jump instructions, and also immediately following a BRcc, a BBIT0, or a BBIT1 instruction.

Pseudo Code

```
FlushPipe()                          /* BRK_S, BRK */
DEBUG[BH] = 1
DEBUG[H] = 1
Halt()
```

Assembly Code Example

A breakpoint instruction may be inserted into any position.

```
MOV   r0, 0x04
ADD   r1, r0, r0
XOR.F 0, r1, 0x8
BRK_S                                ; <---- break here
SUB   r2, r0, 0x3
ADD.NZr1, r0, r0
JZ.D  [r8]
OR    r5, r4, 0x10
```

Breakpoints may be inserted at a jump delay slot.

```
MOV   r0, 0x04
ADD   r1, r0, r0
XOR.F 0, r1, 0x8
SUB   r2, r0, 0x3
ADD.NZr1, r0, r0
JZ.D  [r8]
BRK_S                                ; <---- break here
RLC    r2,r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

	Instruction Code
BRK	00100101011011111000RRRRRR111111
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 Table 8-7 and Table 8-8.

		Instruction Code
BSET<.f>	a,b,c	00100bbb00001111FBBBCCCCCCAAAAAA
BSET<.f>	a,b,u6	00100bbb01001111FBBBuuuuuuAAAAAA
BSET<.f>	b,b,s12	00100bbb10001111FBBBssssssSSSSSS
BSET<.cc><.f>	b,b,c	00100bbb11001111FBBBCCCCCC0QQQQQ
BSET<.cc><.f>	b,b,u6	00100bbb11001111FBBBuuuuuu1QQQQQ
BSET_S	b,b,u5	10111bbb100uuuuu

BSETL

Function

Bit Set Long

Extension Group

Baseline

Operation

if (cc) a = (b | (1<<c));

Instruction Format

op a, b, c

Syntax Example

BSETL<.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 64-bit 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 6 bits of c are used as the bit value. The result is written into the 64-bit destination register (dest). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                              /* BSETL */
 dest = src1 OR (1 << (src2 & 63))
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
BSETL r1,r2,r3   ; Set bit r3 of r2 and write result into r1
```

Syntax and Encoding

		Instruction Code
BSETL<.f>	a,b,c	01011bbb00001111FBBBCCCCCCAAAAAA
BSETL<.f>	a,b,u6	01011bbb01001111FBBBuuuuuuAAAAAA
BSETL<.f>	b,b,s12	01011bbb10001111FBBBssssssSSSSSS
BSETL<.cc><.f>	b,b,c	01011bbb11001111FBBBCCCCCC0QQQQQ
BSETL<.cc><.f>	b,b,u6	01011bbb11001111FBBBuuuuuu1QQQQQ

BTST

Function

Bit Test

Extension Group

Baseline

Operation

if (cc) (b & (1 << c))

Instruction Format

op b, c

Syntax Example

BTST<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logically AND source operand 1 (b) with a bit mask specified by source operand 2 (c). Source operand 2 (c) explicitly defines the bit that is tested in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The flags are updated to reflect the result. The flag setting field, F, is always encoded as 1 for this instruction.

There is no result write-back.

Note BTST and BTST_S always set the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                    /* BTST */
 alu = src1 AND (1 << (src2 & 31))
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[31]
```

Assembly Code Example

```
BTST r2,28    ; Test bit 28 of r2 and update flags on result
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BTST	b,c	00100bbb00010001 1BBBCCCCCCRRRRRR
BTST	b,u6	00100bbb01010001 1BBBuuuuuuRRRRRR
BTST	b,s12	00100bbb10010001 1BBBssssssSSSSSS
BTST<.cc>	b,c	00100bbb11010001 1BBBCCCCCC0QQQQQ
BTST<.cc>	b,u6	00100bbb11010001 1BBBuuuuuu1QQQQQ
BTST_S	b,u5	10111bbb111uuuuu

BTSTL

Function

Bit Test Long

Extension Group

Baseline

Operation

if (cc) (b & (1 << c))

Instruction Format

op b, c

Syntax Example

BTSTL<.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 the 64-bit source operand 1 (b) with a 64-bit mask in which the bit specified by source operand 2 (c) is set to 1 and all other bits are zero. Source operand 2 (c) therefore defines a single bit that is tested in source operand 1 (b). Only the bottom 6 bits of c are used as the bit value. The flags are updated to reflect the result. The flag setting field, F, is always encoded as 1 for this instruction.

There is no result write-back.

BTSTL always sets the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                              /* BTSTL */
 res = src1 AND (1 << (src2 & 63))
 Z_flag = if res==0 then 1 else 0
 N_flag = res[63]
```

Assembly Code Example

```
BTSTL r2,28   ; Test bit 28 of r2 and update flags on result
```

Syntax and Encoding

		Instruction Code
BTSTL	b,c	01011bbb00010001 1BBBCCCCCCRRRRRR
BTSTL	b,u6	01011bbb01010001 1BBBuuuuuuRRRRRR
BTSTL	b,s12	01011bbb10010001 1BBBssssssSSSSSS
BTSTL<.cc>	b,c	01011bbb11010001 1BBBCCCCCC0QQQQQ
BTSTL<.cc>	b,u6	01011bbb11010001 1BBBuuuuuu1QQQQQ

BXOR

Function

Bit Exclusive OR (Bit Toggle)

Extension Group

Baseline

Operation

if (cc) a = (b ^ (1 << c));

Instruction Format

op a, b, c

Syntax Example

BXOR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logically XOR source operand 1 (b) with a bit mask specified by source operand 2 (c). Source operand 2 (c) explicitly defines the bit that is to be toggled in source operand 1 (b). Only the bottom 5 bits of c are used as the bit value. The result is written to the destination register (a). Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                          /* BXOR */
 dest = src1 XOR (1 << (src2 & 31))
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
BXOR r1,r2,r3     ; Toggle bit r3 of r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
BXOR<.f>	a,b,c	00100bbb00010010FBBBCCCCCCAAAAAA
BXOR<.f>	a,b,u6	00100bbb01010010FBBBuuuuuuAAAAAA
BXOR<.f>	b,b,s12	00100bbb10010010FBBBssssssSSSSSS
BXOR<.cc><.f>	b,b,c	00100bbb11010010FBBBCCCCCC0QQQQQ
BXOR<.cc><.f>	b,b,u6	00100bbb11010010FBBBuuuuuu1QQQQQ

BXORL

Function

Bit Exclusive OR Long (Bit Toggle)

Extension Group

Baseline

Operation

if (cc) a = (b ^ (1 << c));

Instruction Format

op a, b, c

Syntax Example

BXORL<.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 the 64-bit source operand 1 (b) with a 64-bit mask, in which the bit specified by source operand 2 (c) is set to 1 and all other bits are 0. Source operand 2 (c) therefore defines the bit that is to be toggled in source operand 1 (b). Only the bottom 6 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                          /* BXORL */
 dest = src1 XOR (1 << (src2 & 63))
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
BXORL r1,r2,56    ; Toggle bit 56 of r2 and write result into r1
```

Syntax and Encoding

		Instruction Code
BXORL<.f>	a,b,c	01011bbb00010010FBBBCCCCCCAAAAAA
BXORL<.f>	a,b,u6	01011bbb01010010FBBBuuuuuuAAAAAA
BXORL<.f>	b,b,s12	01011bbb10010010FBBBssssssSSSSSS
BXORL<.cc><.f>	b,b,c	01011bbb11010010FBBBCCCCCC0QQQQQ
BXORL<.cc><.f>	b,b,u6	01011bbb11010010FBBBuuuuuu1QQQQQ

CLRI

Function

CLRI forces the STATUS32.IE bit to `0`, disabling interrupts.

Extension Group

HAS_INTERRUPTS == 1

Operation

{dest ← { 26'd0, 1'b1, STATUS32.IE, STATUS32.E[3:0] }; STATUS32.IE ← 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.

Pseudo Code

```
{                                                                    /* CLRI */
  dst <-{ 26'd0,  1'b1,  STATUS32.IE,  STATUS32.E[3:0] }
  STATUS32.IE <- 0
}
```

Assembly Code Example

```
CLRI R0     ;clear interrupts and capture interrupt state in R0
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
CLRI	c	00100111001011110000ccccccc111111
CLRI	u6	00100111011011110000uuuuuu111111

If the destination register value is 62, or if a u6 operand is specified, CLRI does not save the interrupt state to the destination register value.

Note Use of limm operands with the CLRI instruction is deprecated. If you use a limm operand with the CLRI instruction, an Illegal Instruction exception is raised.

If a destination C register, whose encoded value is not 62, is specified, CLRI saves the interrupt state from the STATUS32 register to the destination register before disabling interrupts. The resulting destination register can be subsequently used as an operand to the SETI instruction to restore the interrupt enable and interrupt priority level to their values prior to execution of the CLRI instruction. The 32-bit destination register is assigned the following values:

dst[31:6]	dst[5]	**dst[4]**	**dst[3:0]**
26'd0	1	STATUS32.IE	STATUS32.E[3:0]

CMP

Function

Comparison

Extension Group

Baseline

Operation

if (cc) b – c;

Instruction Format

op b, c

Syntax Example

CMP<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

A comparison is performed by subtracting source operand 2 (c) from source operand 1 (b) and subsequently updating the flags.

The CMP instruction always sets the flags even though there is no associated flag setting suffix.

There is no destination register. Therefore, the result of the subtraction is discarded.

Pseudo Code

```
if cc==true then                    /* CMP */
 alu = src1 - src2
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[31]
 C_flag = Carry()
 V_flag = Overflow()
```

Assembly Code Example

```
CMP r1,r2    ; Subtract r2 from r1 and set the flags on the result
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

CMP and CMP_S always set the flags even though there is no associated flag setting suffix.

		Instruction Code
CMP	b,c	00100bbb00011001BBBCCCCCCRRRRRR
CMP	b,u6	00100bbb01011001BBBuuuuuuRRRRRR
CMP	b,s12	00100bbb10011001BBBssssssSSSSSS
CMP<.cc>	b,c	00100bbb11011001BBBCCCCCC0QQQQQ
CMP<.cc>	b,u6	00100bbb11011001BBBuuuuuu1QQQQQ
CMP_S	b,h	01110bbbhhh100HH
CMP_S	h,s3	01110sssshhh101HH
CMP_S	b,u7	11100bbb1uuuuuuu

CMPL

Function

Compare Long

Extension Group

Baseline

Operation

if (cc) b – c;

Instruction Format

op b, c

Syntax Example

CMPL<.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 64-bit source operand 2 (c) from 64-bit source operand 1 (b) and subsequently updating the flags.

The CMPL 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                    /* CMPL */
 alu = src1 - src2
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[63]
 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

CMPL always sets the flags even though there is no associated flag setting suffix.

		Instruction Code
CMPL	b,c	01011bbb00011001BBBCCCCCCRRRRRR
CMPL	b,u6	01011bbb01011001BBBuuuuuuRRRRRR
CMPL	b,s12	01011bbb10011001BBBssssssSSSSSS
CMPL<.cc>	b,c	01011bbb11011001BBBCCCCCC0QQQQQ
CMPL<.cc>	b,u6	01011bbb11011001BBBuuuuuu1QQQQQ

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). The DBNZ instruction performs its decrement operation at 64-bit precision. The branch offset calculation (like all branch offset calculations in ARC64) is also performed at 64-bit precision.

This instruction computes its branch test in parallel with the decrement operation by comparing against 1 instead of 0. This means the wider (and slightly slower) decrement does not appear on the critical path for determining the branch outcome.

This instruction supports only the REG-S12 operand format, as it always requires one register and a signed 12-bit literal operand. Any use of other operand formats will raise an Illegal Instruction exceptions. The register operand must be a writable general-purpose register, i.e. it cannot be a long-immediate indicator (r62), nor can it be the PCL register (r63). Any attempt to use one of those register operands will also raise an Illegal Instruction exception.

The 13-bit literal operand (s13) is encoded in the DBNZ instruction format as an s12 field in which the least-significant bit of the displacement has been omitted. This is because the displacement is always

half-word aligned. The 13-bit relative displacement is added to the current 32-bit word-aligned PC value (PCL) to determine the branch target address.

When DBNZ is used to implement a loop-closing branch the target address represents the address of the first instruction of loop body.

The DBNZ instruction may optionally specify that a delay-slot instruction follows. The delay slot modes, .d, are described in Table 8-13.

Caution The DBNZ instruction cannot appear in the delay slot of another branch or jump instruction.

The processor raises an Illegal Instruction Sequence exception if an executed delay slot contains any of the following:

- Another jump or branch instruction
- 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 Table 8-7 and Table 8-8.

		Instruction Code
DBNZ<.d>	b,s13	00100bbb10 00110N0 BBBssssssSSSSSS

DIV

Function

2's Complement Integer Divide.

Extension Group

Baseline

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 A, "Implementation-dependent Behavior".

Pseudo Code

```
if (cc == true) {                                                    /* DIV */
   if ((src2 != 0) && ((src1 != 0x80000000) || src2 != xffffffff)))
    {
        dest = src1 / src2;
        if (F == 1)  {
            Z = (dest == 0) ? 1 : 0;
            N = dest[31];
            V = 0;
        }
   } else {
      if ((src2 == 0) && (STATUS32.DZ == 1))
         RaiseException (EV_DivZero);
      else {
         /* optional implementation-dependent assignment to dest */
         if (F == 1){
             V = 1;
         }
      }
    }
}
```

Assembly Code Example

```
DIV r1,r2,r3                          ; r1 is assigned r2 / r3
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
DIV<.f>	a,b,c	00101bbb00000100FBBBCCCCCCAAAAAA
DIV<.f>	a,b,u6	00101bbb01000100FBBBuuuuuuAAAAAA
DIV<.f>	b,b,s12	00101bbb10000100FBBBssssssSSSSSS
DIV<.cc><.f>	b,b,c	00101bbb11000100FBBBCCCCCC0QQQQQ
DIV<.cc><.f>	b,b,u6	00101bbb11000100FBBBuuuuuu1QQQQQ

DIVL

Function

2's Complement Integer Divide Long

Extension Group

Baseline

Operation

if (cc) then a = b / c;

Instruction Format

op a, b, c

Syntax Example

DIVL <.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 64-bit divisor (c) is non-zero, the 64-bit destination register is assigned the quotient obtained by signed integer division of 64-bit 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 64 bits. This overflow occurs only in one specific case, which is the division of the largest representable negative value (0x80000000_00000000) 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 64 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, refer to Appendix A, "Implementation-dependent Behavior".

Pseudo Code

```
if (cc == true) {                                                    /* DIVL
   if ((src2 != 0) && (    (src1 != 0x80000000_00000000)             */
                        || (src2 != xffffffff_ffffffff)))
    {
        dest = src1 / src2;
        if (F == 1)  {
            Z = (dest == 0) ? 1 : 0;
            N = dest[63];
            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

```
DIVL r1,r2,r3                        ; r1 is assigned r2 / r3
```

Syntax and Encoding

		Instruction Code
DIVL<.f>	a,b,c	01011bbb00100100FBBBCCCCCCAAAAAA
DIVL<.f>	a,b,u6	01011bbb01100100FBBBuuuuuuAAAAAA
DIVL<.f>	b,b,s12	01011bbb10100100FBBBssssssSSSSSS
DIVL<.cc><.f>	b,b,c	01011bbb11100100FBBBCCCCCC0QQQQQ
DIVL<.cc><.f>	b,b,u6	01011bbb11100100FBBBuuuuuu1QQQQQ

DIVU

Function

Unsigned Integer Divide.

Extension Group

Baseline

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 A, "Implementation-dependent Behavior".

If the flag-update bit is set, an unsigned DIVU operation always clears the N-flag.

Pseudo Code

```
if (cc == true) {                                                    /* DIVU */
   if (src2 != 0) {
        dest = src1 / src2;
        if (F == 1)  {
            Z = (dest == 0) ? 1 : 0;
            N = 0;
            V = 0;
        }
   } else {
      if (STATUS32.DZ == 1)
         RaiseException (EV_DivZero);
      else {
      /* optional implementation-dependent assignment to dest */
      if (F == 1){
             V = 1;
         }
      }
   }
}
```

Assembly Code Example

```
DIVU r1,r2,r3                        ; r1 is assigned r2 / r3
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
DIVU<.f>	a,b,c	00101bbb00000101FBBBCCCCCCAAAAAA
DIVU<.f>	a,b,u6	00101bbb01000101FBBBuuuuuuAAAAAA
DIVU<.f>	b,b,s12	00101bbb10000101FBBBssssssSSSSSS
DIVU<.cc><.f>	b,b,c	00101bbb11000101FBBBCCCCCC0QQQQQ
DIVU<.cc><.f>	b,b,u6	00101bbb11000101FBBBuuuuuu1QQQQQ

DIVUL

Function

Unsigned Integer Divide Long

Extension Group

Baseline

Operation

if (cc) then a = b / c;

Instruction Format

op a, b, c

Syntax Example

DIVUL <.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 64-bit divisor (c) is non-zero, the 64-bit destination register is assigned the quotient obtained by unsigned integer division of 64-bit 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, refer to Appendix A, "Implementation-dependent Behavior".

If the flag-update bit is set, an unsigned DIVUL operation always clears the N-flag.

Pseudo Code

```
if (cc == true) {                                                    /* DIVUL */
   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

```
DIVUL r1,r2,r3                      ; r1 is assigned r2 / r3
```

Syntax and Encoding

		Instruction Code
DIVUL<.f>	a,b,c	01011bbb00100101FBBBCCCCCCAAAAAA
DIVUL<.f>	a,b,u6	01011bbb01100101FBBBuuuuuuAAAAAA
DIVUL<.f>	b,b,s12	01011bbb10100101FBBBssssssSSSSSS
DIVUL<.cc><.f>	b,b,c	01011bbb11100101FBBBCCCCCC0QQQQQ
DIVUL<.cc><.f>	b,b,u6	01011bbb11100101FBBBuuuuuu1QQQQQ

DMACH

This section describes the standard DMACH instruction. .

Function

Dual 16x16 signed integer multiply and accumulate.

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
DMACH<.f>	a,b,c	00101bbb00010010FBBBCCCCCCAAAAAA
DMACH<.f>	a,b,u6	00101bbb01010010FBBBuuuuuuAAAAAA
DMACH<.f>	b,b,s12	00101bbb10010010FBBBssssssSSSSSS
DMACH<.cc><.f>	b,b,c	00101bbb11010010FBBBCCCCCC0QQQQQ
DMACH<.cc><.f>	b,b,u6	00101bbb11010010FBBBuuuuuu1QQQQQ

DMACHU

This section describes the standard DMACHU instruction.

Function

Dual unsigned integer 16x16 multiply and accumulate

Extension Groups

Baseline

Operation

```
if (cc) {
  result = acc + ((b.h0 * c.h0) + (b.h1 * c.h1));
  a = result.w0;
  acc = result;
}
```

`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) {                                                        /* DMACHU */
  result = acc + (b.h0 * c.h0) + (b.h1 * c.h1);
  a = result.w0;
  acc = result;
}
```

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 Table 8-7 and Table 8-8.

		Instruction Code
DMACHU<.f>	a,b,c	00101bbb00010011FBBBCCCCCCAAAAAA
DMACHU<.f>	a,b,u6	00101bbb01010011FBBBuuuuuuAAAAAA
DMACHU<.f>	b,b,s12	00101bbb10010011FBBBssssssSSSSSS
DMACHU<.cc><.f>	b,b,c	00101bbb11010011FBBBCCCCCC0QQQQQ
DMACHU<.cc><.f>	b,b,u6	00101bbb11010011FBBBuuuuuu1QQQQQ

DMACWH

Function

Dual 32x16 multiply and accumulate. .

Extension Group

Baseline

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. `wo` (lower) and `w1` (upper) represent the 32-bit elements of a 64-bit operand.

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 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) {                                                /* 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 Table 8-7 and Table 8-8.

		Instruction Code
DMACWH<.f>	a,b,c	00101bbb00110110FBBBCCCCCCAAAAAA
DMACWH<.f>	a,b,u6	00101bbb01110110FBBBuuuuuuAAAAAA
DMACWH<.f>	b,b,s12	00101bbb10110110FBBBssssssSSSSSS
DMACWH<.cc><.f>	b,b,c	00101bbb11110110FBBBCCCCCC0QQQQQ
DMACWH<.cc><.f>	b,b,u6	00101bbb11110110FBBBuuuuuu1QQQQQ

DMACWHU

Function

Dual unsigned 32x16 multiply and accumulate.

Extension Group

Baseline

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. `wo` (lower) and `w1` (upper) represent the 32-bit elements of a 64-bit operand. A and B are 64-bit operands 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 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) {                                                    /* DMACWHU */
  result= acc + ((B.w0 * c.h0)+(B.w1 * c.h1));
  A = result;
  acc = result;
}
```

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

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 Table 8-7 and Table 8-8.

		Instruction Code
DMACWHU<.f>	a,b,c	00101bbb00110111FBBBCCCCCCAAAAAA
DMACWHU<.f>	a,b,u6	00101bbb01110111FBBBuuuuuuAAAAAA
DMACWHU<.f>	b,b,s12	00101bbb10110111FBBBssssssSSSSSS
DMACWHU<.cc><.f>	b,b,c	00101bbb11110111FBBBCCCCCC0QQQQQ
DMACWHU<.cc><.f>	b,b,u6	00101bbb11110111FBBBuuuuuu1QQQQQ

DMPYH

This section describes the standard DMPYH instruction.

Function

Sum of dual 16x16 multiplication

Extension Group

Baseline

Operation

```
if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}
```

Note `h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. `wo` (lower) represents the 32-bit element of an operand. 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 Table 8-7 and Table 8-8.

		Instruction Code
DMPYH<.f>	a,b,c	00101bbb00010000FBBBCCCCCCAAAAAA
DMPYH<.f>	a,b,u6	00101bbb01010000FBBBuuuuuuAAAAAA
DMPYH<.f>	b,b,s12	00101bbb10010000FBBBssssssSSSSSS
DMPYH<.cc><.f>	b,b,c	00101bbb11010000FBBBCCCCCC0QQQQQ
DMPYH<.cc><.f>	b,b,u6	00101bbb11010000FBBBuuuuuu1QQQQQ

DMPYHU

This section describes the standard DMPYHU instruction.

Function

Sum of dual unsigned 16x16 multiplication

Extension Group

Baseline

Operation

```
if (cc) {result = ((b.h0 * c.h0) + (b.h1 * c.h1)); a = result.w0; acc = result;}
```

Note `h0` (lower) and `h1` (upper) represent the 16-bit elements of a 32-bit operand. `wo` (lower) represents the 32-bit element of an operand. 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) {                                                    /* DMPYHU */
   result = (b.h0 * c.h0) + (b.h1 * c.h1);
   a = result.w0 ;
   acc = result;
}
```

Assembly Code Example

```
DMPYHU r1,r2,r3   ; sum of dual 16x16 unsigned multiply of r2 and r3
```

Syntax and Encoding

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 Table 8-7 and Table 8-8.

		Instruction Code
DMPYHU<.f>	a,b,c	00101bbb00010001FBBBCCCCCCAAAAAA
DMPYHU<.f>	a,b,u6	00101bbb01010001FBBBuuuuuuAAAAAA
DMPYHU<.f>	b,b,s12	00101bbb10010001FBBBssssssSSSSSS
DMPYHU<.cc><.f>	b,b,c	00101bbb11010001FBBBCCCCCC0QQQQQ
DMPYHU<.cc><.f>	b,b,u6	00101bbb11010001FBBBuuuuuu1QQQQQ

DMPYWH

Function

Sum of dual 32x16 multiplication.

Extension Group

Baseline

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. `wo` (lower) and `w1` (upper) represent the 32-bit elements of a 64-bit operand. A and B are 64-bit operands. 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 64-bit accumulator .

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) {                                                /* DMPYWH */
  result =(B.w0 * c.h0)+(B.w1 * c.h1);
  A = result;
  acc = result;
}
```

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 Table 8-7 and Table 8-8.

		Instruction Code
DMPYWH<.f>	a,b,c	00101bbb00110010FBBBCCCCCCAAAAAA
DMPYWH<.f>	a,b,u6	00101bbb01110010FBBBuuuuuuAAAAAA
DMPYWH<.f>	b,b,s12	00101bbb10110010FBBBssssssSSSSSS
DMPYWH<.cc><.f>	b,b,c	00101bbb11110010FBBBCCCCCC0QQQQQ
DMPYWH<.cc><.f>	b,b,u6	00101bbb11110010FBBBuuuuuu1QQQQQ

DMPYWHU

Function

Sum of dual unsigned 32x16 multiplication.

Extension Group

Baseline

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. `wo` (lower) and `w1` (upper) represent the 32-bit elements of a 64-bit operand. A and B are 64-bit operands.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. 64-bit accumulator .

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 Table 8-7 and Table 8-8.

		Instruction Code
DMPYWHU<.f>	a,b,c	00101bbb00110011FBBBCCCCCCAAAAAA
DMPYWHU<.f>	a,b,u6	00101bbb01110011FBBBuuuuuuAAAAAA
DMPYWHU<.f>	b,b,s12	00101bbb10110011FBBBssssssSSSSSS
DMPYWHU<.cc><.f>	b,b,c	00101bbb11110011FBBBCCCCCC0QQQQQ
DMPYWHU<.cc><.f>	b,b,u6	00101bbb11110011FBBBuuuuuu1QQQQQ

DSYNC

Function

Data transfer synchronization instruction

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

	Instruction Code
DSYNC	00100010011011111000 01RRRRRR111111

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 Table 8-7 and Table 8-8.

		Instruction Code
DMB	u3	00100011011011111000 1RRRuuu111111

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

ENTER

Function

Function prolog sequence

Extension Group

ARCv3 Floating-point extension

Operation

Push a collection of registers on to the stack, update stack pointer and frame pointer, as required

Instruction Format

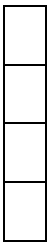
op u12

Syntax Example

ENTER {r14-r20, f26-f31, fp, blink}

STATUS32 Flags Affected

None



Description

This instruction executes the function prolog code, and is typically used by functions that save both integer and floating-point callee-saved registers to the stack. A 12-bit literal operand value defines the set of registers saved to the stack. On completion of this instruction, the stack pointer (r28) is decremented by the total number of bytes required to save the registers. If the frame pointer semantics are specified by the operand value, the stack pointer is copied to the frame pointer on completion of the operation.

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 instruction is not allowed to appear in the delay slot of a jump or branch instruction. The processor raises an Illegal Instruction Sequence exception.

Note An illegal instruction sequence exception is not raised when the ENTER_S instruction has zero operands.

The operand encoding is as follows:

u[3:0]	Indicates the number of general-purpose registers to be saved on the stack, starting from r14 and counting contiguously upwards. If 32 general-purpose registers are configured, at most 14 registers (r14 to r27) can be saved. If u[3:0] > 14an 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). It is possible to save r27 twice; firstly as one of the callee-saved registers specified by the u[3:0] field, and secondly by setting the u[4] bit. This use case is not illegal but serves no useful purpose.
u[5]	If set to 1, the blink register (r31) is saved.
u[6]	**Reserved**. This bit is ignored by the ENTER instruction, but should be set to 0 by the assembler to ensure future compatibility.
u[10:7]	This field defines S, which must be one less than the number of floating-point callee-saved registers saved by this instruction. This 4-bit field lets the ENTER instruction specify between one and 16 floating-point callee-saved registers. When restoring N = S+1 floating-point registers in a configuration with F floating-point registers, contiguously numbered registers from C to C+S are restored, where the u[15:11] field gives C. However, if C+N > F, an Illegal Instruction exception is raised, the instruction attempts to save more floating-point registers than exist. This situation can occur only when the configuration has eight floating-point registers (it cannot arise when there are 16 or 32 floating-point registers).
u[15:11]	This field defines C, which is the identity of the lowest-numbered floating-point register to be saved by the ENTER instruction.

Pseudo Code

```
#define ST2_OP ((DATA_SIZE == 64) ? “stdl” : “std” )
#define ST1_OP ((DATA_SIZE == 64) ? “stl” : “st” )
#define MOV_OP ((DATA_SIZE == 64) ? “movl” : “mov” )
#define SUB_OP ((DATA_SIZE == 64) ? “subl” : “sub” )
#define RDBL_OPTION ((DATA_SIZE == 64) ? M128_OPTION : LL64_OPTION )
#define FST2_OP ((FPR_WIDTH == 64) ? “fstd64” : “fst64” )
#define FST1_OP ((FPR_WIDTH == 64) ? “fst64” : “fst32” )
```

```
#define FDBL_OPTION ((FPR_WIDTH == 64) ? M128_OPTION
                                       : (LL64_OPTION || (DATA_SIZE == 64)))
#if (HAS_FPU == 0)
RaiseException (IllegalInstruction);
#endif
const uint8 max_saved_gprs = 14;
const uint8 first_saved_gpr = 14;
const uint8 gpr_size = DATA_SIZE/8;
const uint8 fpr_size = FPR_WIDTH/8;
if ((u[3:0] > max_saved_gprs) || (u[10:7]+u[15:11]+1 > NUM_FP_REGS))
 RaiseException (IllegalInstruction);
if (InDelaySlot)
 RaiseException (IllegalInstructionSequence);
sint12 num_saved_gprs = u[3:0] + u[4] + u[5];
sint12 num_saved_fprs = u[10:7] + 1;
sint12 first_saved_fpr = u[15:11];
sint12 num_fpr_pairs = num_saved_fprs/2;
sint12 first_fpr_pair = first_saved_fpr
                          + num_saved_fprs
                          - (2*num_fpr_pairs);

sint12 gpr_bytes = gpr_size * num_saved_gprs;
sint12 fpr_bytes = fpr_size * num_saved_fprs;
sint12 frame_bytes = gpr_bytes + fpr_bytes;
sint12 reg_offset = -frame_bytes;
if (u[4] == 1) {
  asm("%s fp,[sp,%d]", ST1_OP, reg_offset);
reg_offset += gpr_size;
}
if (u[5] == 1) {
 asm("%s blink,[sp,%d]", ST1_OP, reg_offset);
 reg_offset += gpr_size;
```

```
}
#if (RDBL_OPTION == 0)
for (I = 0; I < u[3:0]; i++, reg_offset += gpr_size)
 asm("%s r%d,[sp,%d]", ST1_OP, i+first_saved_gpr, reg_offset);
#else
for (I = 0; I < u[3:0]/2; i++, reg_offset += (2*gpr_size))
 asm("%s r%d,[sp,%d]", ST2_OP, (2*i)+first_saved_gpr, reg_offset);
if (u[0] == 1) {
 asm("%s r%d,[sp,%d]", ST1_OP, u[3:0]+first_saved_gpr, reg_offset);
reg_offset += gpr_size;
}
#endif

#if (FDBL_OPTION == 0)
for (I = 0; I < num_saved_fprs; i++, reg_offset += fpr_size)
asm("%s f%d,[sp,%d]", FST1_OP, i+first_saved_fpr, reg_offset);
#else
if (u[0] == 1) {
asm("%s f%d,[sp,%d]", FST1_OP, first_saved_fpr, reg_offset);
reg_offset += fpr_size;
}
for (I = 0; I < num_fpr_pairs; i++, reg_offset += (2*fpr_size))
asm("%s f%d,[sp,%d]", FST2_OP, (2*i)+first_fpr_pair, reg_offset);
#endif
asm("%s sp,sp,%d", SUB_OP, frame_bytes);
if (u[4] == 1)
asm("%s fp,sp", MOV_OP);
```

Assembly Code Example

```
ENTER {r14-r20, f26-f31, fp, blink}
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
ENTER	u16	00100rrU10101110uuuuuuuuuuuuuuu

ENTER_S

Function

Function Prolog Sequence

Extension Group

Baseline

Operation

Push a collection of registers on to the stack, update stack pointer, and frame pointer, as required

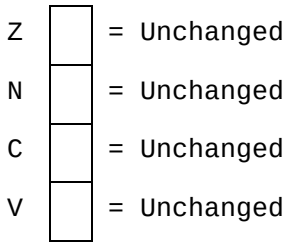
Instruction Format

op u6

Syntax Example

ENTER_S {r14-r27, fp, blink}

STATUS32 Flags Affected



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 8S, 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.

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 r14 and counting contiguously upwards. If 32 general-purpose registers are configured, at most 14 registers (r14 to r27) can be saved. If only 16 general-purpose registers are configured, at most 3 registers (r14 to r16) 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). The frame-pointer is also restored as u[3:0] and through the u[4].
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: store-long (STL) instructions..

Pseudo Code

```
#define ST2_OP     ( (DATA_SIZE == 64) ? "stdl"      : "std"       )
#define ST1_OP     ( (DATA_SIZE == 64) ? "stl"       : "st"        )
#define MOV_OP     ( (DATA_SIZE == 64) ? "movl"      : "mov"       )
#define SUB_OP     ( (DATA_SIZE == 64) ? "subl"      : "sub"       )
#define DBL_OPTION ( (DATA_SIZE == 64) ? M128_OPTION : LL64_OPTION )
  const uint8 max_saved_gprs     = u[4] ? 13: 14;
  const uint8 first_saved_gpr    = 14;
  const uint8 gpr_size           = DATA_SIZE/8;
  if (u[3:0] > max_saved_gprs)
    RaiseException (IllegalInstruction);
  if (InDelaySlot)
    RaiseException (IllegalInstructionSequence);
  sint12 num_saved_gprs   = u[3:0] + u[4] + u[5];
  sint12 frame_bytes      = gpr_size * num_saved_gprs;
  sint12 reg_offset       = -frame_bytes;
  if (u[4] == 1) {
    asm("%s fp,[sp,%d]", ST1_OP, reg_offset);
    reg_offset += gpr_size;
  }
  if (u[5] == 1) {
    asm("%s blink,[sp,%d]", ST1_OP, reg_offset);
    reg_offset += gpr_size;
  }
#if (DBL_OPTION == 0)
  for (I = 0; I < u[3:0]; i++, reg_offset += gpr_size)
    asm("%s r%d,[sp,%d]", ST1_OP, i+first_saved_gpr, reg_offset);
#else
  for (I = 0; I < u[3:0]/2; i++, reg_offset += (2*gpr_size))
    asm("%s r%d,[sp,%d]", ST2_OP, (2*i)+first_saved_gpr, reg_offset);
  if (u[0] == 1) {
    asm("%s r%d,[sp,%d]", ST1_OP, u[3:0]+first_saved_gpr, reg_offset);
    reg_offset += gpr_size;
  }
#endif
  asm("%s sp,sp,%d", SUB_OP, frame_bytes);
  if (u[4] == 1)
    asm("%s fp,sp", MOV_OP);
```

Assembly Code Example

```
ENTER_S {r14-r27, fp, blink}   ;Push registers r14-r27, 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 Table 8-7 and Table 8-8.

		Instruction Code
ENTER_S	u6	110000UU111uuuu0

EX

Function

Atomic Exchange

Extension Group

Baseline

Operation

temp = b ; b = *c; *c = temp;

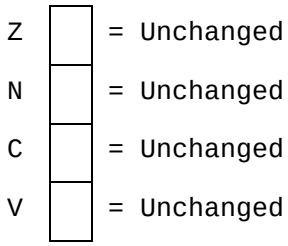
Instruction Format

op b, c

Syntax Example

EX<.di> b,[c]

STATUS32 Flags Affected



Description

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

Two forms are provided: an uncached form (using the .DI directive) for synchronization between multiple processors in systems without cache-coherent shared memory, and a cached form for synchronization between processes on a single-processor system, or on a multiprocessor system with cache-coherent shared memory.

The EX instruction exchanges the contents of the specified memory location with the contents of the specified register. This operation is atomic and the memory system ensures that the memory read and memory write cannot be separated by interrupts or by memory accesses from another source. The memory location must be 32-bit aligned (address bits [1:0] = 0). Thus, the EX-related accesses to data cache never cross a cache line boundary. Even if the data memory alignment checks are disabled (STATUS32.AD bit is set to 1), an EX instruction to an unaligned memory address always raises an EV_Misaligned (Misaligned Data Access) exception.

The status flags are not updated with this instruction.

An immediate value is not permitted to be the destination of the exchange instruction. Using the long immediate indicator in the destination field, B=0x3E, raises a Illegal Instruction exception.

Note	When used in conjunction with an MMU or MPU, both the read and write permissions must be set for EX to operate without causing a protection violation exception.

Pseudo Code

```
int32_t                                                          /* 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 11-1 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 11-2 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 Table 8-7 and Table 8-8.

		Instruction Code
EX<.di>	b,[c]	00100bbb00101111 1BBBCCCCCC001100
EX<.di>	b,[u6]	00100bbb01101111 1BBBuuuuuu001100
EX<.aq.rl>	b,[c]	00100bbb00101111 0BBBCCCCCC001100
EX<.aq.rl>	b,[u6]	00100bbb01101111 0BBBuuuuuu001100

EXL

Function

Atomic Exchange

Extension Group

Baseline

Operation

temp = b ; b = *c; *c = temp;

Instruction Format

op b, c

Syntax Example

EXL<.di> b,[c]

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

An atomic exchange operation, EXL, 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 EXL 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 64-bit aligned (address bits [2:0] = 0). Thus, the EXL-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 EXL instruction to an unaligned memory address always raises an EV_Misaligned (Misaligned Data Access) exception.

The status flags are not updated with this instruction.

An immediate value is not permitted to be the destination of the exchange instruction. Using the long immediate indicator in the destination field, B=0x3E, raises a Illegal Instruction exception.

Note When used in conjunction with an MMU or MPU, both the read and write permissions must be set for EX to operate without causing a protection violation exception.

Pseudo Code

```
int32_t                                                          /* EXL */
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 11-3 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 11-4 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 Table 8-7 and Table 8-8.

		Instruction Code
EXL	b,[c]	01011bbb00101111 0BBBCCCCCC001100
EXL<.aq.rl>	b,[c]	01011bbb00101111 1BBBCCCCCC001100
EXL<.aq.rl>	b,limm	01011bbb01101111 1BBBCCCCCC001100

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 Table 8-7 and Table 8-8.

		Instruction Code
EXTB<.f>	b,c	00100bbb00101111FBBBCCCCCC000111
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 Table 8-7 and Table 8-8.

FFS

		Instruction Code
EXTH<.f>	b,c	00100bbb00101111FBBBCCCCCC001000
EXTH<.f>	b,u6	00100bbb01101111FBBBuuuuuu001000
EXTH_S	b,c	01111bbbccc10000

Function

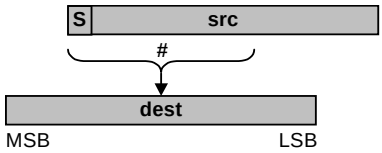
Find First Set

Extension Group

Baseline

Operation

dest ← bit position of last 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
0x------8	3	0	0
:	:	:	:
0x40000000	30	0	0
0x80000000	31	0	1
0x00000000	31	1	0

Pseudo Code

```
s = 31                                              /* 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 and write
            result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FFS<.f>	b,c	00101bbb00101111FBBBCCCCCC010010
FFS<.f>	b,u6	00101bbb01101111FBBBuuuuuu010010

FFSL

Function

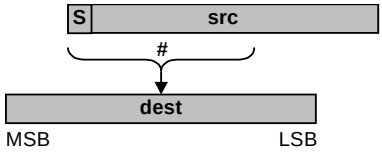
Find First Set

Extension Group

Baseline

Operation

dest ← bit position of last 1 in source operand, from bit 0 upwards



Instruction Format

op b,c

Syntax Example

FFSL<.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
:	:	:	:
0x4000000000000000	62	0	0
0x8000000000000000	63	0	1
0x0000000000000000	64	1	0

Pseudo Code

```
s = 31                                                  /* FFSL */

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[63];
}
```

Assembly Code Example

```
FFSL r1,r2    ;  Find least-significant non-zero bit in r2 and write
              ;result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FFSL<.f>	b,c	00101bbb00101111FBBBCCCCCC010010
FFSL<.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 executed until all flag updates have taken 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

```
if ((src[0] == 1) && (STATUS32[7] == 0)) then                    /* FLAG */
   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                                  ; Setthe 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 Table 8-7 and Table 8-8.

		Instruction Code
FLAG	c	00100rrr00101001 0RRRCCCCCCRRRRRR
FLAG	u6	00100rrr01101001 0RRRuuuuuuRRRRRR
FLAG	s12	00100rrr10101001 0RRRssssssSSSSSS
FLAG<.cc>	c	00100rrr11101001 0RRRCCCCCC0QQQQQ
FLAG<.cc>	u6	00100rrr11101001 0RRRuuuuuu1QQQQQ

FLS

Function

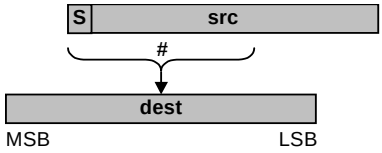
Find Last Set

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
FLS<.f>	b,c	00101bbb00101111FBBBCCCCCC010011
FLS<.f>	b,u6	00101bbb01101111FBBBuuuuuu010011

FLSL

Function

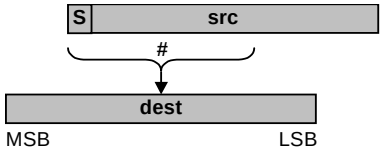
Find Last Set Long

Extension Group

Baseline

Operation

dest ← bit position of last 1 in source operand, from bit 0 upwards



Instruction Format

op b,c

Syntax Example

FLSL<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if source is zero
N	•	= Set if most-significant bit of source is set
C		= Unchanged
V		= Unchanged

Description

Returns the bit position of the most-significant non-zero bit in the source operand. The returned value for a source operand of zero is 0. Examples of returned values are shown in the following table. The '-' symbol signifies any hexadecimal digit:

Operand Value	Result	Z	N
0x8- - - - - - - - - - - - - -	63	0	1
0x4- - - - - - - - - - - - - -	62	0	0
0x2- - - - - - - - - - - - - -	61	0	0

Operand Value	Result	Z	N
:	:	:	:
0x0000000000000004	2	0	0
0x0000000000000002	1	0	0
0x0000000000000001	0	0	0
0x0000000000000000	0	1	0

Pseudo Code

```
s = 63                                                  /* FLSL */

while (src && (((src >> s) & 1) == 0))
 --s;

dest = s;

if (F == 1) then {
  Z_flag = (src == 0) ? 1 : 0;
  N_flag = src[63];
}
```

Assembly Code Example

```
FLSL r1,r2     ; Find most-significant non-zero bit in r2
               ; write result into r1
```

Syntax and Encoding

		Instruction Code
FLSL<.f>	b,c	01011bbb00101111FBBBCCCCCC010011
FLSL<.f>	b,u6	01011bbb01101111FBBBuuuuuu010011

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.

The J and J_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D, or BBITn.D instruction.

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:

Table 11-4 Delay Slot Modes for the Jump and Link Instructions

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 8-13 provides more information about the delay slot modes, .d.

Table 8-9 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
- Indexed jump or execute instructions (JLI_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 Table 8-7 and Table 8-8.

		Instruction Code
J	[c]	00100RRR00100000RRRRCCCCCCRRRRRR
J	u6	00100RRR01100000RRRRuuuuuuRRRRRR
J	s12	00100RRR10100000RRRRssssssSSSSSS

```
J.D     [c]       00100RRR00100001RRRRCCCCCCRRRRRR
J.D     u6        00100RRR01100001RRRRuuuuuuRRRRRR
J.D     s12       00100RRR10100001RRRRssssssSSSSSS
J_S     [b]       01111bbb00000000
J_S.D   [b]       01111bbb00100000
J_S     [BLINK]   01111110111100000
J_S.D   [BLINK]   01111111111100000
```

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. 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:

Table 11-5 Delay Slot Modes for the Jump and Link Instructions

Delay Slot Mode	Description
JEQ_S	Only execute next instruction when *not* branching
JNE_S	Only execute next instruction when *not* branching
Jcc.D	Always execute next instruction

Table 8-13 provides more information about the delay slot modes, .d.

Table 8-9 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
- 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 Table 8-7 and Table 8-8.

		Instruction Code
Jcc	[c]	00100RRR11100000RRRRCCCCCC0QQQQQ
Jcc	u6	00100RRR11100000RRRRuuuuuu1QQQQQ
Jcc.D	[c]	00100RRR11100001RRRRCCCCCC0QQQQQ

```
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.

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:

Table 11-6 Delay Slot Modes for the Jump and Link Instructions

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 8-13 provides more information about the delay slot modes, .d.

Table 8-9 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
- Indexed jump or execute instructions (JLI_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 Table 8-7 and Table 8-8.

		Instruction Code
JL	[c]	00100RRR00100010RRRRCCCCCCRRRRRR
JL	u6	00100RRR01100010RRRRuuuuuuRRRRRR

```
JL       s12    00100RRR10100010RRRRssssssSSSSSS
JL.D     [c]    00100RRR00100011RRRRCCCCCCRRRRRR
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.

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 11-7 Delay Slot Modes for the Jump and Link Instructions

Delay Slot Mode	Description
JLcc	Only execute next instruction when *not* 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 8-13 provides more information about the delay slot modes, .d.

Table 8-9 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
- 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 Table 8-7 and Table 8-8.

Instruction Code

```
JLcc     [c]    00100RRR11100010RRRRCCCCCC0QQQQQ
JLcc     u6     00100RRR11100010RRRRuuuuuu1QQQQQ
JLcc.D   [c]    00100RRR11100011RRRRCCCCCC0QQQQQ
JLcc.D   u6     00100RRR11100011RRRRuuuuuu1QQQQQ
```

JLI_S

Function

Jump and Link Indexed

Extension Group

Baseline

Operation

BLINK = next_PC; PC = JLI_BASE + (u10 << 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 Table 8-7 and Table 8-8.

	Instruction Code
JLI_S u10	01010UUUUUUU1uuu

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 executed until all flag updates have taken 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 Table 8-7 and Table 8-8.

		Instruction Code
KFLAG	c	00100RRR00101001 1RRRCCCCCCRRRRRR
KFLAG	u6	00100RRR01101001 1RRRuuuuuuRRRRRR
KFLAG	s12	00100RRR10101001 1RRRssssssSSSSSS
KFLAG<.cc>	c	00100RRR11101001 1RRRCCCCCC0QQQQQ
KFLAG<.cc>	u6	00100RRR11101001 1RRRuuuuuu1QQQQQ

LDB

Function

Load byte from Memory

Extension Group

Baseline:

Operation

addr=b+c; a = *addr;

Instruction Format

op a, b, c

Syntax Example

LDB<.x><.aa><.di> a,[b,c]

STATUS32 Flags Affected

None

Description

A one-byte memory load occurs from the address given by the sum of the first source operand (b) and optionally a second source operand (c).

The LDB.X instruction sign-extends the 8-bit load data to the 64-bit register width, whereas the LDB instruction zero-extends the load data. The extended value is written into the destination register (a) unless the destination operand specifies a null result by using a LIMM encoding. A load with a null destination register effectively performs a prefetch operation.

Instructions that specify the .DI modifier will bypass any cache present in the system and load directly from memory.

All load-byte instructions may use the .AB, or the .AW addressing modes, but the .AS mode is not supported. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src1
else
  addr = src1 + src2
if ((<.aa> == .AB) or (<.aa> == .AW))
  src1 = src1 + src2

DEBUG[LD] = 1
data = MemoryRead(addr, 1)          /* read 1 byte from memory */
if (<.x> == .X)
  reg[dst] = Sign_Extend64(data, 7) /* copy bit 7 to bits 8 thru 63 */
else
  reg[dst] = Zero_Extend64(data, 8) /* insert 0 into bits 8 thru 63 */

if (NoFurtherLoadsPending())
  DEBUG[LD] = 0
```

Assembly Code Example

```
LDB r0,[r1,4]   ; Load word from memory address r1+4 and write result to r0
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LDB<.x><.aa><.di>	a,[b]	00010bbb000000000BBBDaa01XAAAAAA
LDB<.x><.aa><.di>	a,[b,s9]	00010bbbssssssssSBBBDaa01XAAAAAA
LDB<.x><.aa><.di>	a,[b,c]	00100bbbaa110ZZXDBBBCCCCCCAAAAAA
LDB_S	a,[b,c]	01100bbbccc01aaa
LDB_S	b,[SP,u7]	11000bbb001uuuuu
LDB_S	c,[b,u7]	10000bbbcccuuuuu
LDB_S	R0,[GP,s9]	1100101sssssssss

X ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 01	LDB	LDB.DI	LDB.AW	LDB.AW.DI	LDB.AB	LDB.AB.DI	Illegal	Illegal

1 01	LDB.X	LDB.X.DI	LDB.X.AW	LDB.X.AW.DI	LDB.X.AB	LDB.X.AB.DI	Illegal	Illegal

Note Compact encodings of stack-pointer relative loads have their offset encoded without the least-significant two bits, as all stack addresses are expected to be four-byte aligned. The encoded offset is shifted left by two bit positions creating an offset in which the lower two bits are always 0.

LDH

Function

Load half-word from Memory

Extension Group

Baseline:

Operation

addr=b+c; a = *addr;

Instruction Format

op a, b, c

Syntax Example

LDH<.x><.aa><.di> a,[b,c]

STATUS32 Flags Affected

None

Description

A two-byte (half-word) memory load occurs from the address given by the sum of the first source operand (b) and optionally a second source operand (c), which may be optionally scaled by a factor of 2 when the .AS addressing mode is selected.

The LDH.X instruction sign-extends the 16-bit load data to the 64-bit register width, whereas the LDH instruction zero-extends the load data. The resulting extended value is written into the destination register (a) unless the destination operand specifies a null result by using a LIMM encoding. A load with a null destination register effectively performs a prefetch operation.

Instructions that specify the .DI modifier will bypass any cache present in the system and load directly from memory.

All load half-word instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src1
else if (<.aa> == .AS)
  addr = src1 + (src2 << 1)
else
  addr = src1 + src2

if ((<.aa> == .AB) or (<.aa> == .AW))
  src1 = src1 + src2

DEBUG[LD] = 1
data = MemoryRead(addr, 2)           /* read 2 bytes from memory */
if (<.x> == .X)
  reg[dst] = Sign_Extend64(data, 15) /* copy bit 15 to bits 16 thru 63 */
else
  reg[dst] = Zero_Extend64(data, 16) /* insert 0 into bits 16 thru 63 */
if (NoFurtherLoadsPending())
  DEBUG[LD] = 0
```

Assembly Code Example

```
LDH r0,[r1,4]   ; Load halfword from memory address r1+4 and write result to r0
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LDH<.x><.aa><.di>	a,[b]	00010bbb00000000 0BBBDaa10XAAAAAA
LDH<.x><.aa><.di>	a,[b,s9]	00010bbbssssssss SBBBDaa10XAAAAAA
LDH<.x><.aa><.di>	a,[b,c]	00100bbbaa11010XD BBBCCCCCCAAAAAA
LDH_S	a,[b,c]	01100bbbccc10aaa
LDH_S	c,[b,u6]	10010bbbcccuuuuu
LDH_S	R0,[GP,s10]	11001 10sssssssss
LDH_S.X	c,[b,u6]	10011bbbcccuuuuu

X ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 10	LDH	LDG.DI	LDH.AW	LDH.AW.DI	LDF.AB	LDH.AB.DI	LDH.AS	LDH.AS.DI
1 10	LDH.X	LDH.X.DI	LDH.X.AW	LDH.X.AW.DI	LDH.X.AB	LDH.X.AB.DI	LDH.X.AS	LDH.X.AS.DI

Note LDH_S and LDH_S.X instructions with a constant offset omit the least-significant bit of the constant from the encoded offset, as these are assumed to be zero. The encoded constant is shifted left by one bit positions creating an offset in which the least-significant bit is always 0.

Note For more information on the rules governing non-aligned memory references, see Data Layout in Memory

LD

Function

Load word from Memory

Extension Group

Baseline

Operation

addr=b+c; a = *addr;

Instruction Format

op a, b, c

Syntax Example

LD<.x><.aa><.di> a,[b,c]

STATUS32 Flags Affected

None

Description

A four-byte (word) memory load occurs from the address given by the sum of the first source operand (b) and optionally a second source operand (c), which may be optionally scaled by a factor of 4 when the .AS addressing mode is selected.

The LD.X instruction sign-extends the 32-bit load data to the 64-bit register width, whereas the LD instruction zero-extends the load data. The resulting extended value is written into the destination register (a) unless the destination operand specifies a null result by using a LIMM encoding. A load with a null destination register effectively performs a prefetch operation.

Instructions that specify the .DI modifier will bypass any cache present in the system and load directly from memory.

All load-word instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src1
else if (<.aa> == .AS)
  addr = src1 + (src2 << 2)
else
  addr = src1 + src2

if ((<.aa> == .AB) or (<.aa> == .AW))
  src1 = src1 + src2

DEBUG[LD] = 1
data = MemoryRead(addr, 4)       /* read 4 bytes from memory */
if (<.x> == .X)
  dest = Sign_Extend64(data, 31) /* copy bit 15 to bits 32 thru 63 */
else
  dest = Zero_Extend64(data, 16) /* insert 0 to bits 32 thru 63 */
if (NoFurtherLoadsPending())
  DEBUG[LD] = 0
```

Assembly Code Example

```
LD r0,[r1,4]    ; Load word from memory address r1+4 and write result to r0
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LD<.aa><.di>	a,[b]	00010bbb00000000 0BBBDaa000AAAAAA
LD.X<.aa>	a,[b]	00010bbb00000000 0BBBDaa001AAAAAA
LD<.aa><.di>	a,[b,s9]	00010bbbssssssss SBBBDaa000AAAAAA
LD.X<.aa>	a,[b,s9]	00010bbbssssssss SBBBDaa001AAAAAA
LD<.aa><.di>	a,[b,c]	00100bbbaa110000 DBBBCCCCCCAAAAAA
LD.X<.aa>	a,[b,c]	00100bbbaa110001 DBBBCCCCCCAAAAAA
LD_S	a,[b,c]	01100bbbccc00aaa
LD_S	b,[SP,u7]	10010bbb000uuuuu
LD_S	b,[PCL,u10]	11001bbbuuuuuuuu
LD_S	c,[b,u7]	10011bbbcccuuuuu

```
LD_S        R0,[GP,s11]   11001 00 sssssssss
LD_S        R1,[GP,s11]   01010 SSSSS 00 sss
LD_S.AS     a,[b,c]       01001 bbb ccc 00 aaa
```

X ZZ encoding	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 00	LD	LD.DI	LD.AW	LD.AW.DI	LD.AB	LD.AB.DI	LD.AS	LD.AS.DI

X ZZ encoding	aa D encodings			
	00 0	01 0	10 0	11 0
1 00	LD.X	LD.X.AW	LD.X.AB	LD.X.AS

Note LD_S instructions with a constant offset omit the least-significant two bits of the constant from the encoded offset, as these are assumed to be zero. The encoded constant is shifted left by two bit positions creating an offset in which the least-significant two bits are always 0.

Note For more information on the rules governing non-aligned memory references, see Data Layout in Memory

LDL

Function

Load long-word from Memory

Extension Group

Baseline

Operation

addr=b+c; a = *addr;

Instruction Format

op A, b, c

Syntax Example

LDL<.aa><.di> A,[b,c]

STATUS32 Flags Affected

None

Description

An eight-byte (long word) memory load occurs from the address given by the sum of the first source operand (b) and optionally a second source operand (c), which may be optionally scaled by a factor of 8 when the .AS addressing mode is selected.

The resulting load data is written to the destination register unless the destination operand specifies a null result by using a LIMM encoding. A load with a null destination register effectively performs a prefetch operation.

LDL instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src1
else if (<.aa> == .AS)
  addr = src1 + (src2 << 3)
else
  addr = src1 + src2

if ((<.aa> == .AB) or (<.aa> == .AW))
  src1 = src1 + src2

DEBUG[LD] = 1
reg[dst]   = MemoryRead(addr, 8)      /* read 8 bytes from memory */
if (NoFurtherLoadsPending())
  DEBUG[LD] = 0
```

Assembly Code Example

```
LDL r0,[r1,4]   ; Load 8 bytes from memory address r1+4 and write to r0
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LDL<.aa>	a,[b]	00010bbb000000000BBB1aa001AAAAAA
LDL<.aa>	a,[b,s9]	00010bbbssssssssSBBB1aa001AAAAAA
LDL<.aa>	a,[b,c]	00100bbbaa1100011BBBCCCCCCAAAAAA

X ZZ encoding	aa D encodings			
	00 1	01 1	10 1	11 1
1 00	LDL	LDL.AW	LDL.AB	LDL.AS

Note For more information on the rules governing non-aligned memory references, see Data Layout in Memory

LDDL

Function

Load double long-word from Memory

Extension Group

M128_OPTION

Operation

addr=b+c; a = *addr;

Instruction Format

op A, b, c

Syntax Example

LDDL<.aa><.di> A,[b,c]

STATUS32 Flags Affected

None

Description

A 16-byte (double long- word) memory load occurs from the address given by the sum of the first source operand (b) and optionally a second source operand (c), which may be optionally scaled by a factor of 8 when the .AS addressing mode is selected.

The resulting load data is written into an even-numbered pair of destination registers (A, A+1) unless the destination operand specifies a null result by using a LIMM encoding. A load with a null destination register effectively performs a prefetch operation. The lower eight bytes from memory are written to register A, the upper eight bytes are written to register A+1. An Illegal Instruction exception is raised if A is an odd-numbered register.

All load double long-word instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src1
else if (<.aa> == .AS)
  addr = src1 + (src2 << 3)
else
  addr = src1 + src2

if ((<.aa> == .AB) or (<.aa> == .AW))
  src1 = src1 + src2

DEBUG[LD] = 1
data = MemoryRead(addr, 16)         /* read 16 bytes from memory */
reg[dst]   = data[63:0]
reg[dst+1] = data[127:64]
if (NoFurtherLoadsPending())
  DEBUG[LD] = 0
```

Assembly Code Example

```
LDDL r0,[r1,4]    ; Load 16 bytes from memory address r1+4 and write to r0 and r1
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LDdL<.aa>	a,[b]	00010bbb000000000BBBD1101XAAAAAA
LDdL<.aa>	a,[b,s9]	00010bbbssssssssSBBBD1101XAAAAAA
LDdL<.aa>	a,[b,c]	00100bbb11111001XDBBBCCCCCCAAAAAA

X ZZ encoding	aa D encodings	
	11 0	11 1
0 01	LDDL	LDDL.AS
1 01	LDD.AW	LDDL.AB

Note For more information on the rules governing non-aligned memory references, see Data Layout in Memory

LEAVE

Function

Function epilog sequence

Extension Group

ARCv3 Floating-point extension

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 u12

Syntax Example

LEAVE {r14-r20, f26-f31, fp, blink, pcl}

STATUS32 Flags Affected

None

Description

This instruction executes the function epilogue code, and is used by functions that wish to restore both integer and floating-point callee-saved registers from the stack. The set of registers restored from the stack is defined by a 12-bit literal operand value. On completion of this instruction, the stack pointer (r28) is incremented by the total number of bytes on the stack occupied by the restored registers. If frame pointer semantics are specified by the operand value, the stack pointer is restored from the frame pointer before any stack values are accessed. 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 instruction must not be 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.

If the stack pointer is not 32-bit aligned, a misaligned access exception is always raised, irrespective of the STATUS32.AD bit.

The operand encoding is as follows:

u[3:0]	Indicates the number of general-purpose registers to be restored from the stack, starting from r14 and counting contiguously upwards. When 32 general-purpose registers are configured, at most 14 registers (r14 to r27) can be restored. Therefore, if u[3:0] > 14, an Illegal Instruction exception is raised.

u[4]	If set to 1, r28 (stack-pointer) is set to r27 (frame-pointer) before registers are loaded from the stack. The frame-pointer is also restored from the stack. It is possible to restore r27 twice, firstly as one of the callee-saved registers specified by the u[3:0] field, and secondly by setting the u[4] bit. This use case is not illegal but serves no useful purpose.
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.
u[10:7]	This field defines S, which must be one less than the number of floating-point callee-saved registers restored by this instruction. This 4-bit field lets the LEAVE instruction specify between one and 16 floating-point callee-saved registers. When restoring N = S+1 floating-point registers in a configuration with F floating-point registers, contiguously numbered registers from C to C+S are restored, where the u[15:11] field gives C. However, if C+N > F, an Illegal Instruction exception is raised, the instruction attempts to restore more floating-point registers than exist. This situation can occur only when the configuration has eight floating-point registers (it cannot arise when there are 16 or 32 floating-point registers).
u[15:11]	This field defines C, which is the identity of the lowest-numbered floating-point register to be restored by the LEAVE instruction.

Pseudo Code

```
#define ST2_OP ((DATA_SIZE == 64) ? “stdl” : “std” )
#define ST1_OP ((DATA_SIZE == 64) ? “stl” : “st” )
#define MOV_OP ((DATA_SIZE == 64) ? “movl” : “mov” )
#define SUB_OP ((DATA_SIZE == 64) ? “subl” : “sub” )
#define RDBL_OPTION ((DATA_SIZE == 64) ? M128_OPTION : LL64_OPTION )
#define FLD2_OP ((FPR_WIDTH == 64) ? “fldd64” : “fld64” )
#define FLD1_OP ((FPR_WIDTH == 64) ? “fld64” : “fld32” )
#define FDBL_OPTION ((FPR_WIDTH == 64) ? LDL_OPTION
                                       : (LL64_OPTION || (DATA_SIZE == 64)))
#if (HAS_FPU == 0)
     RaiseException (IllegalInstruction);
#endif
     const uint8 max_saved_gprs = 14;
     const uint8 first_saved_gpr = 14;
     const uint8 gpr_size = DATA_SIZE/8;
     const uint8 fpr_size = FPR_WIDTH/8;
if ((u[3:0] > max_saved_gprs) || (u[10:7]+u[15:11]+1 > NUM_FP_REGS))
```

```
    RaiseException (IllegalInstruction);
if (InDelaySlot)
    RaiseException (IllegalInstructionSequence);
sint12 num_saved_gprs = u[3:0] + u[4] + u[5];
sint12 num_saved_fprs = u[10:7] + 1;
sint12 first_saved_fpr = u[15:11];
sint12 num_fpr_pairs = num_saved_fprs/2;
sint12 first_fpr_pair = first_saved_fpr
                        + num_saved_fprs
                        - (2*num_fpr_pairs);
sint12 gpr_bytes = gpr_size * num_saved_gprs;
sint12 fpr_bytes = fpr_size * num_saved_fprs;
sint12 frame_bytes = gpr_bytes + fpr_bytes;
sint12 reg_offset = u[4] * gpr_size;
if (u[4] == 1)
   asm("%s sp,fp", MOV_OP);
if (u[5] == 1) {
  asm("%s blink,[sp,%d]", LD1_OP, reg_offset);
reg_offset += gpr_size;
}
#if (RDBL_OPTION == 0)
for (I = 0; I < u[3:0]; i++, reg_offset += gpr_size)
  asm("%s r%d,[sp,%d]", LD1_OP, i+first_saved_gpr, reg_offset);
#else
for (I = 0; I < u[3:0]/2; i++, reg_offset += (2*gpr_size))
  asm("%s r%d,[sp,%d]", LD2_OP, (2*i)+first_saved_gpr, reg_offset);
if (u[0] == 1) {
  asm("%s r%d,[sp,%d]", LD1_OP, u[3:0]+first_saved_gpr, reg_offset);
reg_offset += gpr_size;
}
#endif
#if (FDBL_OPTION == 0)
```

```
for (I = 0; I< num_saved_fprs; i++, reg_offset += fpr_size)
  asm("%s f%d,[sp,%d]", FLD1_OP, i+first_saved_fpr, reg_offset);
#else
if (u[0] == 1) {
 asm("%s r%d,[sp,%d]", FLD1_OP, first_saved_fpr, reg_offset);
reg_offset += fpr_size;
}
for (I = 0; I < num_fpr_pairs; i++, reg_offset += (2*fpr_size))
 asm("%s f%d,[sp,%d]", FLD2_OP, (2*i)+first_fpr_pair, reg_offset);
#endif
if (u[4] == 1)
 asm("%s fp,[sp]", LD1_OP);
if (u[6] == 1)
 asm("j.d [blink]");
 asm("%s sp,sp,%d ", ADD_OP, frame_bytes);
```

Assembly Code Example

```
LEAVE {r14-r20, f26-f31, fp, blink, pcl}
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LEAVE	u16	00100rrU10101110 1uuuuuuuuuuuuuuu

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 {r14-r27, 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 8S, 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 r14 and counting contiguously upwards. When 32 general-purpose registers are configured, at most 14 registers (r14 to r27) can be restored. When only 16 general-purpose registers are configured, at most 3 registers (r14 to r16) 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 as u[3:0] and through the u[4].
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]
```

64-bit operations are used to load registers, which are equivalent to:

```
LDL r31, [sp,+16]
```

Pseudo Code

```
#define LD2_OP     ( (DATA_SIZE == 64) ? "lddl"      : "ldd"       )
#define LD1_OP     ( (DATA_SIZE == 64) ? "ldl"       : "ld"        )
#define MOV_OP     ( (DATA_SIZE == 64) ? "movl"      : "mov"       )
#define ADD_OP     ( (DATA_SIZE == 64) ? "addl"      : "add"       )
#define DBL_OPTION ( (DATA_SIZE == 64) ? LDL_OPTION  : LL64_OPTION )
  const uint8 max_saved_gprs     = u[4] ? 13: 14;
  const uint8 first_saved_gpr    = 14;
  const uint8 gpr_size           = DATA_SIZE/8;
  if (u[3:0] > max_saved_gprs)
```

```
      RaiseException (IllegalInstruction);
    if (InDelaySlot)
      RaiseException (IllegalInstructionSequence);
    sint12 num_saved_gprs   = u[3:0] + u[4] + u[5];
    sint12 frame_bytes      = gpr_size * num_saved_gprs;
    sint12 reg_offset       = u[4] * gpr_size;
    if (u[4] == 1)
      asm("%s sp,fp", MOV_OP);
    if (u[5] == 1) {
      asm("%s blink,[sp,%d]", LD1_OP, reg_offset);
      reg_offset += gpr_size;
    }
  #if (DBL_OPTION == 0)
    for (I = 0; I < u[3:0]; i++, reg_offset += gpr_size)
      asm("%s r%d,[sp,%d]", LD1_OP, i+first_saved_gpr, reg_offset);
  #else
    for (I = 0; I < u[3:0]/2; i++, reg_offset += (2*gpr_size))
      asm("%s r%d,[sp,%d]", LD2_OP, (2*i)+first_saved_gpr, reg_offset);
    if (u[0] == 1) {
      asm("%s r%d,[sp,%d]", LD1_OP, u[3:0]+first_saved_gpr, reg_offset);
      reg_offset += gpr_size;
    }
  #endif
    if (u[4] == 1)
      asm("%s fp,[sp]", LD1_OP);
    if (u[6] == 1)
      asm("j.d [blink]");
    asm("%s sp,sp,%d ", ADD_OP, frame_bytes);
```

Assembly Code Example

```
LEAVE_S {r14-r27, blink, pcl}       ;Pop registers r14-r27, 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 Table 8-7 and Table 8-8.

		Instruction Code
LEAVE_S	u7	11000UUU110uuuu0

LLOCK

Function

Load Locked

Extension Group

Baseline

Operation

b = _llock(c);

Instruction Format

op b,c

Syntax Example

LLOCK<.di> b, [c]

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

The 32-bit word, located at the address given by the source operand, is loaded from the data address space into the destination register. If the load operation completes, the equivalent physical address is saved in the non-architecturally visible Lock Physical Address (LPA) register, and the non-architecturally visible Lock Flag (LF) register is set to 1.

The LF register is cleared whenever an exception or interrupt is taken, or whenever the processor completes a store instruction to the address contained in the LPA register.

If any external agent, such as another processor, completes a store to the address contained in the LPA register, the LF register is cleared. Such writes, and the clearing of LF, are an atomic action.

In a virtual memory environment, the LLOCK instruction does not check the write permissions on the operand location. Therefore, the following SCOND instruction may fail because of insufficient privileges even if the LLOCK instruction completes its read successfully.

The LLOCK.DI instruction operates directly on external memory, bypassing any data cache that may be present in the path from the processor to the physical memory system. This form must be used in

implementations of the ARCv3 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 ARCv3 that provide hardware cache coherency for shared memory accesses, or in systems with a single processor.

The effective address of LLOCK must be aligned to a 4-byte boundary, otherwise an EV_Misaligned exception is raised even if STATUS32.AD==1.

Pseudo Code

```
EA  = PhysicalAddress(c); translate address from c if
                        ; VM enabled
b   = ReadWord(EA[31:2]); read 32-bit word from memory into b
LPA = EA[31:2]          ; for non-PAE builds; PAE builds use EA[39:2]
LF  = 1
```

Assembly Code Example

```
LLOCK r1,[r2]   ; Load from address given by r2 and set LF to 1 Set
                ;LPA to r2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LLOCK<.di>	b,[c]	00100bbb00101111DBBBCCCCCC010000
LLOCK<.di>	b,[u6]	00100bbb01101111DBBBuuuuuu010000

LLOCKL

Function

Load locked on 64-bit data

Extension Group

Baseline

Operation

_llockl(b,c);

Instruction Format

op B,C

Syntax Example

LLOCKL B, [C]

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

The LLOCKL instruction is also similar to the LLOCKD instruction except that LLOCKL is available only in ARC64, and it loads a 64-bit value into a 64-bit core register.

The effective address of LLOCKL 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

```
LLOCKL r0,[r2]    ; Load from address given by r2 to the 64-bit
                  ; register r0 and set LF to 1. Set LPA to r2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LLOCKL	b,[c]	01011bbb00101111 0BBBCCCCCC010000
LLOCKL<aq>	b,[c]	01011bbb00101111 1BBBCCCCCC010000

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 Table 8-7 and Table 8-8.

		Instruction Code
LR	b,[c]	00100bbb00101010RBBBCCCCCCRRRRRR
LR	b,[u6]	00100bbb01101010RBBBuuuuuuRRRRRR
LR	b,[s12]	00100bbb10101010RBBBssssssSSSSSS

LRL

Function

Load from Auxiliary Register Long

Extension Group

Baseline

Operation

b = _lrl(c);

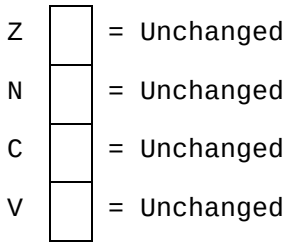
Instruction Format

op b,c

Syntax Example

LRL b,[c]

STATUS32 Flags Affected



Description

Get the data from the auxiliary register whose number is obtained from the source operand (c) and place the resulting 64-bit data value into the 64-bit destination register (b).

The LRL instruction executes unconditionally. Therefore, specifying an operand mode (bits 23:22) of 0x3 raises an Illegal Instruction.

Pseudo Code

```
dest = Aux_reg(src)                /* LRL */
```

Assembly Code Example

```
LRL r1,[r2]       ; Load contents of Aux. register pointed
                  ; to by r2 into 64-bit register 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.

		Instruction Code
LRL	b,[c]	01011bbb00101010RBBBCCCCCCRRRRRR
LRL	b,[u6]	01011bbb01101010RBBBuuuuuuRRRRRR
LRL	b,[s12]	01011bbb10101010RBBBssssssSSSSSS

LSL16

Function

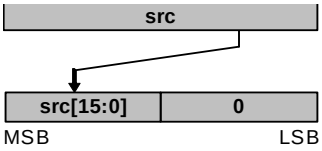
Logical Shift Left 16

Extension Group

Baseline

Operation

b = c << 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 Table 8-7 and Table 8-8.

LSL16<.f>	b,c	00101bbb00101111FBBBCCCCCC001010
LSL16<.f>	b,u6	00101bbb01101111FBBBuuuuuu001010

LSL8

Function

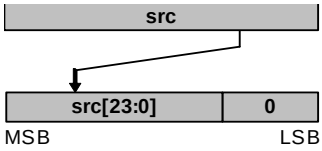
Logical Shift Left 8

Extension Group

Baseline

Operation

b = c << 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 Table 8-7 and Table 8-8.

LSL8<.f>	b,c	00101bbb00101111FBBBCCCCCC001111
LSL8<.f>	b,u6	00101bbb01101111FBBBuuuuuu001111

LSR

Function

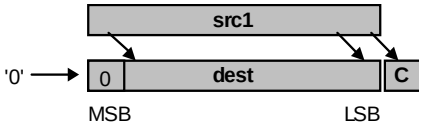
Logical Shift Right

Extension Group

Baseline

Operation

b = (unsigned) c >> 1;



Instruction Format

op b,c

Syntax Example

LSR<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Logically right shift the source operand (c) by one and place the result into the destination register (b).

The most-significant bit of the result is replaced with 0.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 1                                          /* LSR */
dest[31] = 0
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = src[0]
```

Assembly Code Example

```
LSR r1,r2    ; Logical shift right contents of r2 by one bit and write
             ;result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
LSR<.f>	b,c	00100bbb00101111FBBBCCCCCC000010
LSR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000010
LSR_S	b,c	01111bbbccc11101

LSRL

Function

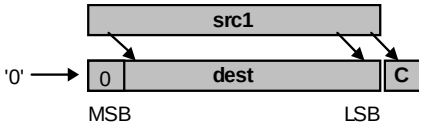
Logical Shift Right Long

Extension Group

Baseline

Operation

b = (unsigned) c >> 1;



Instruction Format

op b,c

Syntax Example

LSRL<.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 64-bit source operand (c) by one and place the result into the 64-bit 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                                          /* LSRL */
dest[63] = 0
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[63]
 C_flag = src[0]
```

Assembly Code Example

```
LSRL r1,r2   ; Logical shift right contents of r2 by one bit and write
             ; result into r1
```

Syntax and Encoding

		Instruction Code
LSRL<.f>	a,b,c	01011bbb00100001FBBBCCCCCCAAAAAA
LSRL<.f>	a,b,u6	01011bbb01100001FBBBuuuuuuAAAAAA
LSRL<.f>	b,b,s12	01011bbb10100001FBBBssssssSSSSSS
LSRL<.f>	b,b,c	01011bbb11100001FBBBCCCCCC0QQQQQ
LSRL<.f>	b,b,u6	01011bbb11100001FBBBuuuuuu1QQQQQ
LSRL<.f>	b,c	01011bbb00101111FBBBCCCCCC000010
LSRL<.f>	b,u6	01011bbb01101111FBBBuuuuuu000010

LSR multiple

Function

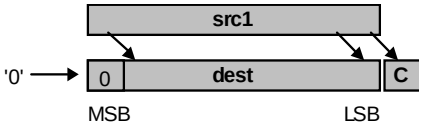
Multiple Logical Shift Right

Extension Group

Baseline

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[sr2-1]
```

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 Table 8-7 and Table 8-8.

		Instruction Code
LSR<.f>	a,b,c	00101bbb00000001FBBBCCCCCCAAAAAA
LSR<.f>	a,b,u6	00101bbb01000001FBBBuuuuuuAAAAAA
LSR<.f>	b,b,s12	00101bbb10000001FBBBssssssSSSSSS
LSR<.cc><.f>	b,b,c	00101bbb11000001FBBBCCCCCC0QQQQQ
LSR<.cc><.f>	b,b,u6	00101bbb11000001FBBBuuuuuu1QQQQQ
LSR_S	b,b,c	01111bbbccc11001
LSR_S	b,b,u5	10111bbb001uuuuu

LSRL Multiple

Function

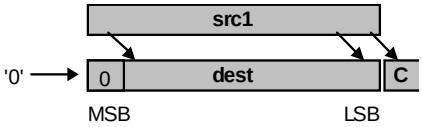
Multiple Logical Shift Right Long

Extension Group

Baseline

Operation

if (cc) a = (unsigned) b >> c;



Instruction Format

op a, b, c

Syntax Example

LSRL<.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 64-bit result in the destination register. Only the bottom 6bits 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                                    /* LSRL Multiple */
 dest = src1 >> (src2 & 63)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = if src2==0 then 0 else src1[sr2-1]
```

Assembly Code Example

```
LSRL r1,r2,r3    ; Logical shift right contents of r2 by r3 bits and
                 ; write result into r1
```

Syntax and Encoding

		Instruction Code
LSRL<.f>	a,b,c	01011bbb00100001FBBBCCCCCCAAAAAA
LSRL<.f>	a,b,u6	01011bbb01100001FBBBuuuuuuAAAAAA
LSRL<.f>	b,b,s12	01011bbb10100001FBBBssssssSSSSSS
LSRL<.cc><.f>	b,b,c	01011bbb11100001FBBBCCCCCC0QQQQQ
LSRL<.cc><.f>	b,b,u6	01011bbb11100001FBBBuuuuuu1QQQQQ

LSR16

Function

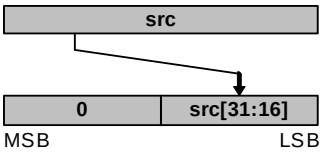
Logical Shift Right 16

Extension Group

Baseline

Operation

b = c >> 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 Table 8-7 and Table 8-8.

```
LSR16<.f>   b,c      00101bbb00101111FBBBCCCCCC001011
LSR16<.f>   b,u6     00101bbb01101111FBBBuuuuuu001011
```

LSR8

Function

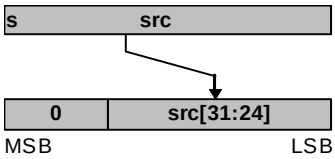
Logical Shift Right 8

Extension Group

Baseline

Operation

b = c >> 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 Table 8-7 and Table 8-8.

LSR8<.f>	b,c	00101bbb00101111FBBBCCCCCC001110
LSR8<.f>	b,u6	00101bbb01101111FBBBuuuuuu001110

MAC

This section describes the standard MAC instruction.

Function

Signed 32x32 multiplication and accumulation.

Extension Group

Baseline

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) {                                                    /* MAC*/
  result = acc +(b * c);
  a = result.w0;
  acc = result;
}
```

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 Table 8-7 and Table 8-8.

		Instruction Code
MAC<.f>	a,b,c	00101bbb00001110FBBBCCCCCCAAAAAA
MAC<.f>	a,b,u6	00101bbb01001110FBBBuuuuuuAAAAAA
MAC<.f>	b,b,s12	00101bbb10001110FBBBssssssSSSSSS
MAC<.cc><.f>	b,b,c	00101bbb11001110FBBBCCCCCC0QQQQQ
MAC<.cc><.f>	b,b,u6	00101bbb11001110FBBBuuuuuu1QQQQQ

MACD

This section describes the standard MACD instruction.

Function

Signed 32x32 multiplication and accumulation. Returns a 64-bit result.

Extension Group

Baseline

Operation

```
if (cc) {
   result = acc + (b * c);
   A = result;
   acc = result;
}
```

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. 64-bit 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) {                                                    /* MACD*/
  result = acc + (b * c);
  A = result;
  acc = result;
}
```

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 Table 8-7 and Table 8-8.

		Instruction Code
MACD<.f>	a,b,c	00101bbb00011010FBBBCCCCCCAAAAAA
MACD<.f>	a,b,u6	00101bbb01011010FBBBuuuuuuAAAAAA
MACD<.f>	b,b,s12	00101bbb10011010FBBBssssssSSSSSS
MACD<.cc><.f>	b,b,c	00101bbb11011010FBBBCCCCCC0QQQQQ
MACD<.cc><.f>	b,b,u6	00101bbb11011010FBBBuuuuuu1QQQQQ

MACDU

This section describes the standard MACDU instruction.

Function

Unsigned 32x32 multiplication and accumulation. Return a 64-bit result.

Extension Group

Baseline

Operation

```
if (cc) {
   result = acc + (b * c);
   A = result;
   acc = result;
}
```

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 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 Table 8-7 and Table 8-8.

		Instruction Code
MACDU<.f>	a,b,c	00101bbb00011011FBBBCCCCCCAAAAAA
MACDU<.f>	a,b,u6	00101bbb01011011FBBBuuuuuuAAAAAA
MACDU<.f>	b,b,s12	00101bbb10011011FBBBssssssSSSSSS
MACDU<.cc><.f>	b,b,c	00101bbb11011011FBBBCCCCCC0QQQQQ
MACDU<.cc><.f>	b,b,u6	00101bbb11011011FBBBuuuuuu1QQQQQ

MACU

Function

Unsigned 32x32 multiplication and accumulation.

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
MACU<.f>	a,b,c	00101bbb00001111FBBBCCCCCCAAAAAA
MACU<.f>	a,b,u6	00101bbb01001111FBBBuuuuuuAAAAAA
MACU<.f>	b,b,s12	00101bbb10001111FBBBssssssSSSSSS
MACU<.cc><.f>	b,b,c	00101bbb11001111FBBBCCCCCC0QQQQQ
MACU<.cc><.f>	b,b,u6	00101bbb11001111FBBBuuuuuu1QQQQQ

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 Table 8-7 and Table 8-8.

		Instruction Code
MAX<.f>	a,b,c	00100bbb00001000FBBBCCCCCCAAAAAA
MAX<.f>	a,b,u6	00100bbb01001000FBBBuuuuuuAAAAAA
MAX<.f>	b,b,s12	00100bbb10001000FBBBssssssSSSSSS
MAX<.cc><.f>	b,b,c	00100bbb11001000FBBBCCCCCC0QQQQQ
MAX<.cc><.f>	b,b,u6	00100bbb11001000FBBBuuuuuu1QQQQQ
MAX<.aq.rl>	b,[c]	00100bbb00101111mBBBCCCCCC110101

MAXL

Function

Return Maximum Value

Extension Group

Baseline

Operation

if (cc) a = (b>c) ? b : c;

Instruction Format

op a, b, c

Syntax Example

MAXL<.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

```
MAXL r1,r2,r3    ; Take maximum of r2 and r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
MAXL<.f>	a,b,c	01011bbb00001000FBBBCCCCCCAAAAAA
MAXL<.f>	a,b,u6	01011bbb01001000FBBBuuuuuuAAAAAA
MAXL<.f>	b,b,s12	01011bbb10001000FBBBssssssSSSSSS
MAXL<.cc><.f>	b,b,c	01011bbb11001000FBBBCCCCCC0QQQQQ
MAXL<.cc><.f>	b,b,u6	01011bbb11001000FBBBuuuuuu1QQQQQ

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 Table 8-7 and Table 8-8.

		Instruction Code
MIN<.f>	a,b,c	00100bbb00001001FBBBCCCCCCAAAAAA
MIN<.f>	a,b,u6	00100bbb01001001FBBBuuuuuuAAAAAA
MIN<.f>	b,b,s12	00100bbb10001001FBBBssssssSSSSSS
MIN<.cc><.f>	b,b,c	00100bbb11001001FBBBCCCCCC0QQQQQ
MIN<.cc><.f>	b,b,u6	00100bbb11001001FBBBuuuuuu1QQQQQ
MIN<.aq.rl>	b,[c]	00100bbb00101111mBBBCCCCCC110100

MINL

Function

Return Minimum Value

Extension Group

Baseline

Operation

if (cc) a = (b<c) ? b : c;

Instruction Format

op a, b, c

Syntax Example

MINL<.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

```
MINL r1,r2,r3    ; Take minimum of r2 and r3 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
MINL<.f>	a,b,c	00100bbb00001001FBBBCCCCCCAAAAAA
MINL<.f>	a,b,u6	00100bbb01001001FBBBuuuuuuAAAAAA
MINL<.f>	b,b,s12	00100bbb10001001FBBBssssssSSSSSS
MINL<.cc><.f>	b,b,c	00100bbb11001001FBBBCCCCCC0QQQQQ
MINL<.cc><.f>	b,b,u6	00100bbb11001001FBBBuuuuuu1QQQQQ

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 least significant 32 bits of 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 Table 8-7 and Table 8-8.

		Instruction Code
MOV<.f>	b,c	00100bbb00001010FBBBCCCCCCRRRRRR
MOV<.f>	b,u6	00100bbb01001010FBBBuuuuuuRRRRRR
MOV<.f>	b,s12	00100bbb10001010FBBBssssssSSSSSS
MOV<.cc><.f>	b,c	00100bbb11001010FBBBCCCCCC0QQQQQ
MOV<.cc><.f>	b,u6	00100bbb11001010FBBBuuuuuu1QQQQQ
MOV_S	h,s3	01110sssshhh011HH
MOV_S	0,s3	01110sss110011 11
MOV_S.NE	b,h	01110bbbhhh111HH
MOV_S.NE	b,limm	01110bbb110111 11
MOV_S	b,u8	11011bbbuuuuuuuu
MOV_S	g,h	01000ggghhhGG0HH
MOV_S	g,limm	01000ggg110GG011
MOV_S	0,h	01000110hhh110HH
MOV_S	0,limm	01000110110110 11

MOVL

Function

Move Long

Extension Group

Baseline

Operation

if (cc) b = c;

Instruction Format

op b, c

Syntax Example

MOVL<.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                          /* MOVL */
 dest = src
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
MOVL r1,r2     ; Move 64-bit contents of r2 into r1
```

Syntax and Encoding

		Instruction Code
MOVL<.f>	b,c	01011bbb00001010FBBBCCCCCCRRRRRR
MOVL<.f>	b,u6	01011bbb01001010FBBBuuuuuuRRRRRR
MOVL<.f>	b,s12	01011bbb10001010FBBBssssssSSSSSS
MOVL<.cc><.f>	b,c	01011bbb11001010FBBBCCCCCC0QQQQQ
MOVL<.cc><.f>	b,u6	01011bbb11001010FBBBuuuuuu1QQQQQ
MOVL_S	g,h	01000gggghhhgg1hh
MOVL_S	b-u8	11011bbbuuuuuuuu

MOVHL

Function

Move to 32-bit high part of long destination

Extension Group

Baseline

Operation

if (cc) b = (c << 32)

Instruction Format

op b, c

Syntax Example

MOVHL<.cc> b,c

MOVHS_S h, LIMM

STATUS32 Flags Affected

None

Description

The least-significant 32 bits of the source operand (c) are moved to the most significant 32 bits of the destination operand (b), and the least significant 32 bits of b are cleared.

Pseudo Code

```
if cc==true {
dest[63:32] = src[31:0]
dest[31:0] = 0
}
```

Assembly Code Example

```
MOVHL_S r1, 0x87654321 ; set r1 to 0x8765432100000000 (limm opd)
MOVHL r0, -4 ; set r1 to 0xfffffffc00000000 (s12 opd)
```

Instruction Encodings

		Instruction Code
MOVHL	b,c	01011bbb00001011 0BBBCCCCCCRRRRRR
MOVHL	b,u6	01011bbb01001011 0BBBuuuuuuRRRRRR
MOVHL	b,s12	01011bbb10001011 0BBBssssssSSSSSS
MOVHL<.cc>	b,c	01011bbb11001011 0BBBCCCCCC0QQQQQ
MOVHL<.cc>	b,u6	01011bbb11001011 0BBBuuuuuu1QQQQQ
MOVHL_S	h, LIMM	01110000hhh010HH

MPY, MPY_S

Function

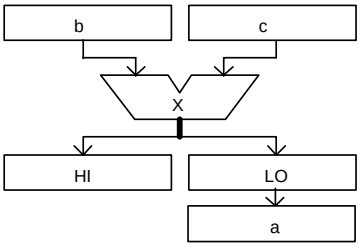
32 x 32 Signed Multiply, returning least-significant 32-bit word of the result.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
MPY<.f>	a,b,c	00100bbb00011010FBBBCCCCCCAAAAAA
MPY<.f>	a,b,u6	00100bbb01011010FBBBuuuuuuAAAAAA
MPY<.f>	b,b,s12	00100bbb10011010FBBBssssssSSSSSS
MPY<.cc><.f>	b,b,c	00100bbb11011010FBBBCCCCCC0QQQQQ
MPY<.cc><.f>	b,b,u6	00100bbb11011010FBBBuuuuuu1QQQQQ
MPY_S	b,b,c	01111bbbccc01100

MPYL

Function

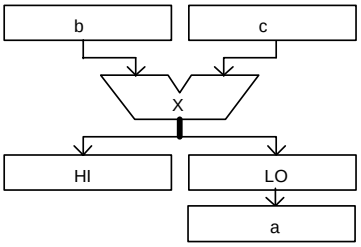
64x 64 Signed Multiply, returning least-significant 64-bit word of the result.

Extension Group

-mpy64==true

Operation

if (cc) a = (signed) (b * c) & 0xFFFF_FFFF;



Instruction Format

op a, b, c

Syntax Example

MPY<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 64-bit result is set
C		= Unchanged
V	•	= Set when the 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 64-bit by 64-bit multiplication of operand 1 and operand 2. Return the least-significant 64 bits of the result in the destination register.

Pseudo Code

```
if cc==true then                                                    /* MPYL */
 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
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
MPYL	a,b,c	01011bbb00110000 0BBBCCCCCCAAAAAA
MPYL	a,b,u6	01011bbb01110000 0BBBuuuuuuAAAAAA
MPYL	b,b,s12	01011bbb10110000 0BBBssssssSSSSSS
MPYL<.cc>	b,b,c	01011bbb11110000 0BBBCCCCCC0QQQQQ
MPYL<.cc>	b,b,u6	01011bbb11110000 0BBBuuuuuu1QQQQQ

MPYD

This section describes the standard MPYD instruction.

Function

Signed 32x32 multiplication. Return a 64-bit result.

Extension Group

Baseline

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

Compute the 32x32 product of the first and second source operands. Assign the 64-bit product to 64-bit accumulator .

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is always cleared, and the sign (N) flag is set to the sign of the result.

Pseudo Code

```
if(cc) {                                                    /* MPYD*/
  result = (b * c);
  A = result;
  acc = result;
}
```

Assembly Code Example

```
MPYD r0,r2,r3    ; 32x32 multiplication of r2 and r3. The result is
                 ; stored in (r1, r0).
```

Syntax and Encoding

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 Table 8-7 and Table 8-8.

		Instruction Code
MPYD<.f>	a,b,c	00101bbb00011000FBBBCCCCCCAAAAAA
MPYD<.f>	a,b,u6	00101bbb01011000FBBBuuuuuuAAAAAA
MPYD<.f>	b,b,s12	00101bbb10011000FBBBssssssSSSSSS
MPYD<.cc><.f>	b,b,c	00101bbb11011000FBBBCCCCCC0QQQQQ
MPYD<.cc><.f>	b,b,u6	00101bbb11011000FBBBuuuuuu1QQQQQ

MPYDU

Function

Unsigned 32x32 multiplication with a 64-bit result.

Extension Group

Baseline

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

Compute the unsigned 32x32 product of the first and second source operands. Assign the unsigned 64-bit product 64-bit accumulator .

Flags are updated only if the set flags suffix (.F) is used. The overflow (V) flag is always cleared.

Pseudo Code

```
if(cc) {                                                    /* MPYDU*/
  result = (b * c);
  A = result;
  acc = result;
}
```

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 Table 8-7 and Table 8-8.

		Instruction Code
MPYDU<.f>	a,b,c	00101bbb00011001FBBBCCCCCCAAAAAA
MPYDU<.f>	a,b,u6	00101bbb01011001FBBBuuuuuuAAAAAA
MPYDU<.f>	b,b,s12	00101bbb10011001FBBBssssssSSSSSS
MPYDU<.cc><.f>	b,b,c	00101bbb11011001FBBBCCCCCC0QQQQQ
MPYDU<.cc><.f>	b,b,u6	00101bbb11011001FBBBuuuuuu1QQQQQ

MPYM MPYH

Function

32 x 32 Signed Multiply High.

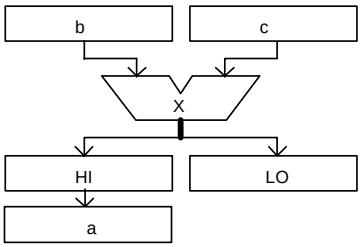
> **Note** The MPYH mnemonic is deprecated.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
MPYM<.f>	a,b,c	00100bbb00011011FBBBCCCCCCAAAAAA
MPYM<.f>	a,b,u6	00100bbb01011011FBBBuuuuuuAAAAAA
MPYM<.f>	b,b,s12	00100bbb10011011FBBBssssssSSSSSS
MPYM<.cc><.f>	b,b,c	00100bbb11011011FBBBCCCCCC0QQQQQ
MPYM<.cc><.f>	b,b,u6	00100bbb11011011FBBBuuuuuu1QQQQQ

MPYMU MPYHU

Function

32 x 32 Unsigned integer Multiply High.

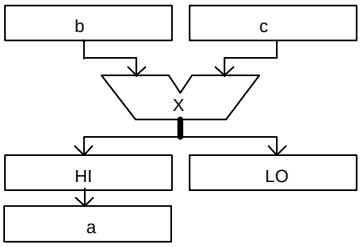
> **Note** The MPYHU mnemonic is deprecated.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
MPYMU<.f>	a,b,c	00100bbb00011100FBBBCCCCCCAAAAAA
MPYMU<.f>	a,b,u6	00100bbb01011100FBBBuuuuuuAAAAAA
MPYMU<.f>	b,b,s12	00100bbb10011100FBBBssssssSSSSSS
MPYMU<.cc><.f>	b,b,c	00100bbb11011100FBBBCCCCCC0QQQQQ
MPYMU<.cc><.f>	b,b,u6	00100bbb11011100FBBBuuuuuu1QQQQQ

MPYML

Function

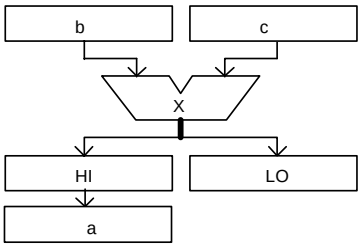
64x 64 Signed Multiply High.

Extension Group

-mpy64==true

Operation

if (cc) a = (signed) (b * c) >> 64;



Instruction Format

op a, b, c

Syntax Example

MPYML<.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 64-bit by 64-bit multiplication of operand 1 and operand 2, placing the most-significant 64 bits of the 128-bit result in the destination register. Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                                    /*MPYML*/
 dest = (src1 * src2) >> 64
```

Assembly Code Example

```
MPYML 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 Table 8-7 and Table 8-8.

		Instruction Code
MPYML<.f>	a,b,c	01011bbb00110001 0BBBCCCCCCAAAAAA
MPYML<.f>	a,b,u6	01011bbb01110001 0BBBuuuuuuAAAAAA
MPYML<.f>	b,b,s12	01011bbb10110001 0BBBssssssSSSSSS
MPYML<.cc><.f>	b,b,c	01011bbb11110001 0BBBCCCCCC0QQQQQ
MPYML<.cc><.f>	b,b,u6	01011bbb11110001 0BBBuuuuuu1QQQQQ

MPYMUL

Function

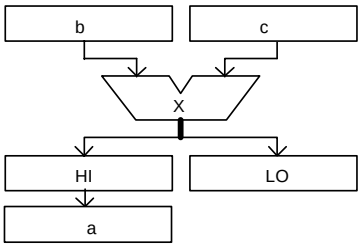
64x 64 Unsigned Multiply High.

Extension Group

-mpy64==true

Operation

if (cc) a = (unsigned) (b * c) >> 64;



Instruction Format

op a, b, c

Syntax Example

MPYMUL<.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 64-bit by 64-bit multiplication of operand 1 and operand 2, placing the most-significant 64 bits of the 128-bit result in the destination register.

Pseudo Code

```
if cc==true then                                    /*MPYMUL*/
 dest = (src1 * src2) >> 64
```

Assembly Code Example

```
MPYMUL 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 Table 8-7 and Table 8-8.

		Instruction Code
MPYMUL<.f>	a,b,c	01011bbb00110010 0BBBCCCCCCAAAAAA
MPYMUL<.f>	a,b,u6	01011bbb01110010 0BBBuuuuuuAAAAAA
MPYMUL<.f>	b,b,s12	01011bbb10110010 0BBBssssssSSSSSS
MPYMUL<.cc><.f>	b,b,c	01011bbb11110010 0BBBCCCCCC0QQQQQ
MPYMUL<.cc><.f>	b,b,u6	01011bbb11110010 0BBBuuuuuu1QQQQQ

MPYMSUL

Function

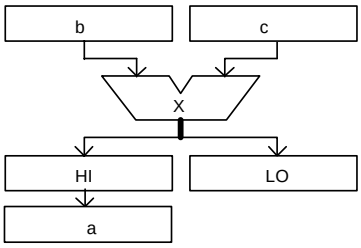
64 x 64 multiplication of unsigned and signed operands, the higher 64-bits of the result are returned.

Extension Group

-mpy64==true

Operation

```
if (cc) a = (unsigned)(b)* (signed)(c) >> 64;
```



Instruction Format

op a, b, c

Syntax Example

MPYMSUL<.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 64-bit by 64-bit multiplication of operand 1 and operand 2, placing the most-significant 64 bits of the 128-bit result in the destination register.

Pseudo Code

```
if cc==true then                                    /*MPYMSUL*/
 dest = (src1 * src2) >> 64
```

Assembly Code Example

```
MPYMSUL          ;Multiply r2 by r3 and put high part of the result in r1
r1,r2,r3
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
MPYMSUL<.f>	a,b,c	01011bbb00110011 0BBBCCCCCCAAAAAA
MPYMSUL<.f>	a,b,u6	01011bbb01110011 0BBBuuuuuuAAAAAA
MPYMSUL<.f>	b,b,s12	01011bbb10110011 0BBBssssssSSSSSS
MPYMSUL<.cc><.f>	b,b,c	01011bbb11110011 0BBBCCCCCC0QQQQQ
MPYMSUL<.cc><.f>	b,b,u6	01011bbb11110011 0BBBuuuuuu1QQQQQ

MPYU

Function

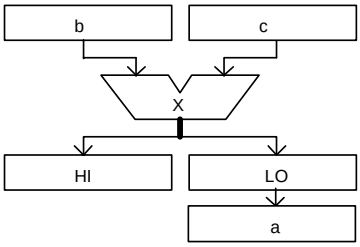
32 x 32 Unsigned Multiply, returning least-significant 32-bit word of result.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
MPYU<.f>	a,b,c	00100bbb00011101FBBBCCCCCCAAAAAA
MPYU<.f>	a,b,u6	00100bbb01011101FBBBuuuuuuAAAAAA
MPYU<.f>	b,b,s12	00100bbb10011101FBBBssssssSSSSSS
MPYU<.cc><.f>	b,b,c	00100bbb11011101FBBBCCCCCC0QQQQQ
MPYU<.cc><.f>	b,b,u6	00100bbb11011101FBBBuuuuuu1QQQQQ

MPYUW, MPYUW_S

Function

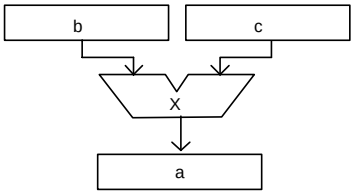
16 x 16 unsigned Multiply, return 32-bit word result.

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
MPYUW<.f>	a,b,c	00100bbb00011111FBBBCCCCCCAAAAAA
MPYUW<.f>	a,b,u6	00100bbb01011111FBBBuuuuuuAAAAAA
MPYUW<.f>	b,b,s12	00100bbb10011111FBBBssssssSSSSSS
MPYUW<.cc><.f>	b,b,c	00100bbb11011111FBBBCCCCCC0QQQQQ
MPYUW<.cc><.f>	b,b,u6	00100bbb11011111FBBBuuuuuu1QQQQQ
MPYUW_S	b,b,c	01111bbbccc01010

MPYW, MPYW_S

Function

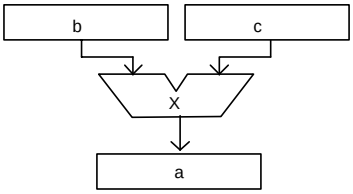
16 x 16 Signed Multiply.

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
MPYW<.f>	a,b,c	00100bbb00011110FBBBCCCCCCAAAAAA
MPYW<.f>	a,b,u6	00100bbb01011110FBBBuuuuuuAAAAAA
MPYW<.f>	b,b,s12	00100bbb10011110FBBBssssssSSSSSS
MPYW<.cc><.f>	b,b,c	00100bbb11011110FBBBCCCCCC0QQQQQ
MPYW<.cc><.f>	b,b,u6	00100bbb11011110FBBBuuuuuu1QQQQQ
MPYW_S	b,b,c	01111bbbccc01001

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. Negation of 64-bit values can be performed with RSUBL.

For more information about the encodings, see Table 8-7 and Table 8-8.

		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. For more information about the encodings, see Table 8-7 and Table 8-8.

	Instruction Code
NOP_S	01111000111 00000
NOP	00100110010010100111000000000000

NORM

Function

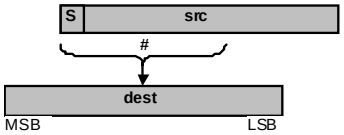
Normalize

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
NORM<.f>	b,c	00101bbb00101111FBBBCCCCCC000001
NORM<.f>	b,u6	00101bbb01101111FBBBuuuuuu000001

NORMH NORMW

Function

Normalize 16-bit Half-word

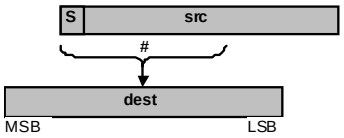
> **Note** The NORMW mnemonic is deprecated.

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
NORMH<.f>	b,c	00101bbb00101111FBBBCCCCCC001000

```
NORMH<.f>   b,u6     00101bbb01101111FBBBuuuuuu001000
```

NORML

Function

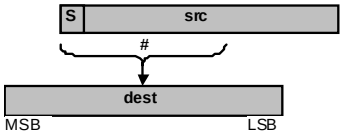
Normalize long

Extension Group

Baseline

Operation

for (i=0; (i<=63) && (c>>i != 0) && (c>>i != -1); i++); b= 63-i;



Instruction Format

op b,c

Syntax Example

NORML<.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 64-bit 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 64-bit signed integer.

Any flag updates occur only if the set flags suffix (.F) is used. The result value for a source operand of zero is 63. Examples of returned values are shown in the following table:

Operand Value	Returned Value
0x0000000000000000	0x000000000000003F
0x0000000000000001	0x000000000000003E
0x1FFFFFFFFFFFFFFF	0x0000000000000002
0x3FFFFFFFFFFFFFFF	0x0000000000000001
0x7FFFFFFFFFFFFFFF	0x0000000000000000
0x8000000000000000	0x0000000000000000
0xC000000000000000	0x0000000000000001
0xE000000000000000	0x0000000000000002
0xFFFFFFFFFFFFFFFF	0x000000000000003F

Pseudo Code

```
for (i=0;i<=63;i++)                    /* NORML */
  if (src>>i ==0) break
  if (src>>i ==-1) break
endfor
dest = 63=i
if F==1 then
  Z_flag = if src==0 then 1 else 0
  N_flag = src[31]
```

Assembly Code Example

```
NORML r1,r2    ; Compute normalization integer for r2, and
               ; write result into r1
```

Syntax and Encoding

		Instruction Code
NORML<.f>	b,c	01011bbb00101111FBBBCCCCCC100001
NORML<.f>	b,u6	01011bbb01101111FBBBuuuuuu100001

NOT

Function

Bitwise NOT

Extension Group

Baseline

Operation

b = ~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 Table 8-7 and Table 8-8.

		Instruction Code
NOT<.f>	b,c	00100bbb00101111FBBBCCCCCC001010
NOT<.f>	b,u6	00100bbb01101111FBBBuuuuuu001010
NOT_S	b,c	01111bbbccc10010

NOTL

Function

Bitwise NOT Long

Extension Group

Baseline

Operation

b = ~c;

Instruction Format

op b,c

Syntax Example

NOTL<.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 64-bit source operand (c) with the result placed into the 64-bit destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = NOT(src)                              /* NOTL */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
NOTL r1,r2   ; Logical bitwise NOT r2 and write result into r1
```

Syntax and Encoding

		Instruction Code
NOTL<.f>	b,c	01011bbb00101111FBBBCCCCCC001010
NOTL<.f>	b,u6	01011bbb01101111FBBBuuuuuu001010

OR

Function

Bitwise OR

Extension Group

Baseline

Operation

if (cc) a = b | c;

Instruction Format

op a, b, c

Syntax Example

OR<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Logical bitwise OR of source operand 1 (b) with source operand 2 (c). The result is written into the destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                        /* OR */
 dest = src1 OR src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
OR r1,r2,r3   ; Logical bitwise OR contents of r2 with r3
              ; and write result into r1;
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
OR<.f>	a,b,c	00100bbb00000101FBBBCCCCCCAAAAAA
OR<.f>	a,b,u6	00100bbb01000101FBBBuuuuuuAAAAAA
OR<.f>	b,b,s12	00100bbb10000101FBBBssssssSSSSSS
OR<.cc><.f>	b,b,c	00100bbb11000101FBBBCCCCCC0QQQQQ
OR<.cc><.f>	b,b,u6	00100bbb11000101FBBBuuuuuu1QQQQQ
OR_S	b,b,c	01111bbbccc00101

ORL

Function

Bitwise OR Long

Extension Group

Baseline

Operation

if (cc) a = b | c;

Instruction Format

op a, b, c

Syntax Example

ORL<.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 64-bit source operand 1 (b) with 64-bit source operand 2 (c). The result is written into the 64-bit destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                              /* ORL */
 dest = src1 OR src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
ORL r1,r2,r3    ; 64-bit logical bitwise OR of r2 with r3
                ; and write result into r1
```

Syntax and Encoding

		Instruction Code
ORL<.f>	a,b,c	01011bbb00000101FBBBCCCCCCAAAAAA
ORL<.f>	a,b,u6	01011bbb01000101FBBBuuuuuuAAAAAA
ORL<.f>	b,b,s12	01011bbb10000101FBBBssssssSSSSSS
ORL<.cc><.f>	b,b,c	01011bbb11000101FBBBCCCCCC0QQQQQ
ORL<.cc><.f>	b,b,u6	01011bbb11000101FBBBuuuuuu1QQQQQ
ORL_S<.cc><.f>	b,b,c	01111bbbccc10111
ORL_S	h,h,limm	01110000hhh110HH
ORL_S	h,PCL,limm	01110010hhh110HH

POPL_S

Function

Pop from Stack

Extension Group

Baseline

Operation

b = *SP; SP=SP+8;

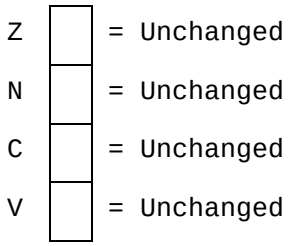
Instruction Format

op b

Syntax Example

POPL_S b

STATUS32 Flags Affected



Description

Perform a 64-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 8-bytes (SP=SP+8).

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:

LDL.AB a, [SP,+8]

Pseudo Code

```
dest = Memory(SP, 8)   /* POPL */
SP   = SP + 8
```

Assembly Code Example

```
POPL_S r1   ; Load word from memory at address SP and write this word to
            ;r1, and then add 8 to SP
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
POPL_S	b	11000bbb1100BBB1

POPDL_S

Function

Pop from Stack

Extension Group

-m128_option is true

Operation

b = *SP; SP=SP+16;

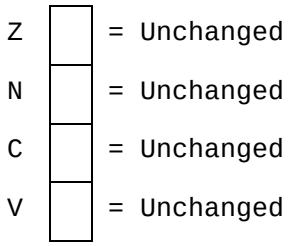
Instruction Format

op b

Syntax Example

POPDL_S b

STATUS32 Flags Affected



Description

Perform a 128-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 16-bytes (SP=SP+16).

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:

LDDL.AB a, [SP,+16]

Pseudo Code

```
dest = Memory(SP, 16)    /* POPDL_S */
SP   = SP + 16
```

Assembly Code Example

```
POPDL_S r1    ; Load word from memory at address SP and write this word to
              ;r1, and then add 16 to SP
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
POPDL_S	b	11000bbb1101BBB1

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

Baseline

Operation

Fetch_with_modified_ownership(L2/external_memory) AND Fill_cacheLine_with_zeros

Instruction Format

op b,c

Syntax Example

PREALLOC<.aa> [b,c]

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

A PREALLOC instruction that misses in the data cache allocates the line in modified state. The line is allocated without the overhead of fetching it from memory. The line is filled with zeros.

A PREALLOC instruction that hits in the data cache or targets any other memory such as peripheral, ICCM, DCCM, or uncached behaves as a NOP.

The PREALLOC instruction does not raise data memory address-related exceptions such as: page fault, memory error, or misaligned exception.

Software must not assume the PREALLOC instruction zeroes out the line, that is the software should explicitly write to the line with store instructions.

Assembly Code Example

```
PREALLOC r1, r2   ; allocates a cache line as a preparation for
                  ; writing the whole line without the overhead
                  ; of fetching the line from memory
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
PREALLOC<.aa>	[b,c]	00100bbbaa110RR10BBBCCCCCC111110
PREALLOC<.aa>	[b,s9]	00010bbbssssssssSBBB0aaRR1111110

PREFETCH

Function

Prefetch from Memory

Extension Group

Baseline

Operation

addr=b+c; temp = *addr;

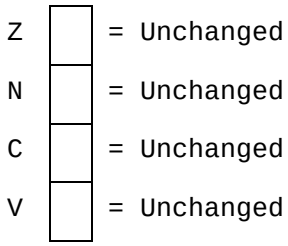
Instruction Format

op b,c

Syntax Example

PREFETCH<.aa> [b,c]

STATUS32 Flags Affected



Description

The PREFETCH instruction is provided as a synonym for a particular encoding of the LD instruction.

- A memory load occurs from the address that is calculated by adding source operand 1 (b) with source operand 2 (c) and the returning load data is loaded into the data cache. The returning load is not written to any core register.
- The actual pre-fetch of data from memory takes place only if the effective address computed by this instruction would not trigger a data memory address-related exception had it been used as the effective address of a load instruction.
- This instruction does not raise data memory address-related exceptions such as: page fault, memory error, misaligned exception.
- PREFETCH and PREFETCHW are always guaranteed to execute, but the pre-fetch aspect of their behavior is no more than a hint, and is not part of the instruction semantics.
- The address update and the increment of the PC always take place. Depending on the address update mode, this instruction can be thought of as either a NOP or ADD.
- The address write-back mode can be selected by use of the <.aa> syntax.

- When using the scaled source addressing mode (.AS), the scale factor is set to 4 for 32-bit Word accesses. The status flags are not updated with this instruction.

Table 8-14 lists the address write-back modes.

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

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

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

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

The <x> and <d> fields specify which particular prefetch-type instruction is encoded, when the destination register is r62 (null destination).

Some combinations of values for <zz>, <aa>, <d>, and <x> are illegal, although this may also depend on whether the instruction is a load or a prefetch-type.

The PREFETCH, PREFETCHW and PREALLOC instructions are encoded using <d> and <x> bits, when the destination register is null (r62), as follows:

Table 11-8 PREFETCH Instruction Encodings

Class	<d>	<x>	Instruction
PREFETCH	0	0	PREFETCH
PREFETCH	0	1	PREALLOC
PREFETCH	1	0	PREFETCHW
PREFETCH	1	1	Reserved (currently deemed illegal)
Load	-	-	LD, LDB, LDH or LDD, depending on <zz>

The following combinations of load or prefetch-type modifier fields are illegal, and raise an Illegal Instruction exception.

- The <x> and <d> bits together select the prefetch-type opcode.
- The '-' character indicates a don't care case.
- The <aa> and <zz> fields operate identically for both prefetch and load instruction classes. Thus, prefetch-type instructions may have both data-size and address pre/post-increment or scaling, just as load instructions.

- The PREFETCHW and PREALLOC instructions are implemented only if the OS_OPT_OPTION is enabled. When this is not the case, any attempt to execute those instructions will be interpreted as a PREFETCH instruction.

All illegal combinations of <d>, <x>, <aa>, and <zz> are shown in Table 11-9. These combinations cover all load and prefetch-type instructions in 32-bit encodings.

Table 11-9 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 Table 8-7 and Table 8-8.

		Instruction Code
PREFETCH<.aa>	[b,c]	00100bbbaa110RR00BBBCCCCCC111110
PREFETCH<.aa>	[b,s9]	00010bbbssssssssSBBB0aaRR0111110

PREFETCHW

Function

Prefetch line from the memory with an intention to write. Move data from the L2 cache or external memory closer to the processor in anticipation of a write

Extension Group

Baseline

Operation

Fetch_with_exclusive_ownership(L2/external_memory)

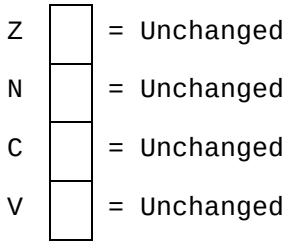
Instruction Format

op b,c

Syntax Example

PREFETCHW<.aa> [b,c]

STATUS32 Flags Affected



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 11-8 on page 661 and Table 11-9 on page 662. For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
PREFETCHW<.aa>	[b,c]	00100bbbaa110RR01BBBCCCCCC111110
PREFETCHW<.aa>	[b,s9]	00010bbbssssssssSBBB1aaRR0111110

PUSHL_S

Function

Push onto Stack

Extension Group

Baseline

Operation

SP=SP-8; *SP = b;

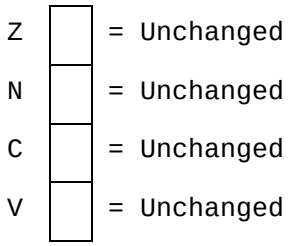
Instruction Format

op b

Syntax Example

PUSHL_S b

STATUS32 Flags Affected



Description

Decrement 8 bytes from the implicit stack pointer address SP (28) and perform a 64-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 STH STB STD STL STDL when used with the following syntax:

STL.AW c,[SP,-8]

Pseudo Code

```
SP = SP - 8           /* PUSHL_S */
Memory(SP, 8) = src
```

Assembly Code Example

```
PUSHL_S r1    ; Subtract 8 from SP and then store the word in r1 to memory
              ;at address SP
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
PUSHL_S	b	11000bbb1110BBB1

PUSHDL_S

Function

Push onto Stack

Extension Group

-m128_option is true

Operation

SP=SP-16; *SP = b;

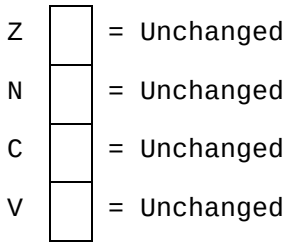
Instruction Format

op b

Syntax Example

PUSHDL_S b

STATUS32 Flags Affected



Description

Decrement 16 bytes from the implicit stack pointer address SP (28) and perform a 128-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 STH STB STD STL STDL when used with the following syntax:

ST.AW c,[SP,-4]

Pseudo Code

```
SP = SP - 16          /* PUSHDL_S */
Memory(SP, 16) =
src
```

Assembly Code Example

```
PUSHDL_S     ; Subtract 16 from SP and then store the word in r1 to
r1           memory ;at address SP
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
PUSHDL_S	b	11000bbb1111BBB1

QMACH

Function

Quad 16x16 multiply and accumulate.

Extension Group

Baseline

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 64-bit operands. 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 64-bit 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 (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 Table 8-7 and Table 8-8.

		Instruction Code
QMACH<.f>	a,b,c	00101bbb00110100FBBBCCCCCCAAAAAA
QMACH<.f>	a,b,u6	00101bbb01110100FBBBuuuuuuAAAAAA
QMACH<.f>	b,b,s12	00101bbb10110100FBBBssssssSSSSSS
QMACH<.cc><.f>	b,b,c	00101bbb11110100FBBBCCCCCC0QQQQQ
QMACH<.cc><.f>	b,b,u6	00101bbb11110100FBBBuuuuuu1QQQQQ

QMACHU

Function

Quad unsigned 16x16 multiply and accumulate.

Extension Group

Baseline

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 64-bit operands. 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 64-bit 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 (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 Table 8-7 and Table 8-8.

		Instruction Code
QMACHU<.f>	a,b,c	00101bbb00110101FBBBCCCCCCAAAAAA
QMACHU<.f>	a,b,u6	00101bbb01110101FBBBuuuuuuAAAAAA
QMACHU<.f>	b,b,s12	00101bbb10110101FBBBssssssSSSSSS
QMACHU<.cc><.f>	b,b,c	00101bbb11110101FBBBCCCCCC0QQQQQ
QMACHU<.cc><.f>	b,b,u6	00101bbb11110101FBBBuuuuuu1QQQQQ

QMPYH

Function

Quad 16x16 multiplication.

Extension Group

Baseline

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 64-bit operands. 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 64-bit accumulator .

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) {                                                                    /* QMPYH*/
   result = (B.h0*C.h0) + (B.h1*C.h1) + (B.h2*C.h2) + (B.h3*C.h3);
   A = result;
   acc = result;
}
```

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 Table 8-7 and Table 8-8.

		Instruction Code
QMPYH<.f>	a,b,c	00101bbb00110000FBBBCCCCCCAAAAAA
QMPYH<.f>	a,b,u6	00101bbb01110000FBBBuuuuuuAAAAAA
QMPYH<.f>	b,b,s12	00101bbb10110000FBBBssssssSSSSSS
QMPYH<.cc><.f>	b,b,c	00101bbb11110000FBBBCCCCCC0QQQQQ
QMPYH<.cc><.f>	b,b,u6	00101bbb11110000FBBBuuuuuu1QQQQQ

QMPYHU

Function

Quad unsigned 16x16 multiply. .

Extension Group

Baseline

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 64-bit operands. 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 64-bit accumulator .

Flags are updated only if the set flags suffix (.F) is used. This instruction always clears the overflow (V) flag.

Pseudo Code

```
if(cc) {                                                                    /* QMPYHU*/
   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 Table 8-7 and Table 8-8.

		Instruction Code
QMPYHU<.f>	a,b,c	00101bbb00110001FBBBCCCCCCAAAAAA
QMPYHU<.f>	a,b,u6	00101bbb01110001FBBBuuuuuuAAAAAA
QMPYHU<.f>	b,b,s12	00101bbb10110001FBBBssssssSSSSSS
QMPYHU<.cc><.f>	b,b,c	00101bbb11110001FBBBCCCCCC0QQQQQ
QMPYHU<.cc><.f>	b,b,u6	00101bbb11110001FBBBuuuuuu1QQQQQ

RCMP

Function

Reverse Comparison

Extension Group

BASELINE

Operation

if (cc) c – b;

Instruction Format

op b,c

Syntax Example

RCMP<.cc> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

A reverse comparison is performed by subtracting source operand 1 (B) from source operand 2 (C) and subsequently updating the flags.

There is no destination register. Therefore, the result of the subtract is discarded.

RCMP always sets the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                    /* RCMP */
 alu = src2 - src1
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[31]
 C_flag = Carry()
 V_flag = Overflow()
```

Assembly Code Example

```
RCMP r1,r2    ; Subtract r1 from r2 and set the flags on the result
```

Syntax and Encoding

RCMP always set the flags even though there is no associated flag setting suffix.

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
RCMP	b,c	00100bbb00011011BBBCCCCCCRRRRRR
RCMP	b,u6	00100bbb01011011BBBuuuuuuRRRRRR
RCMP	b,s12	00100bbb10011011BBBssssssSSSSSS
RCMP<.cc>	b,c	00100bbb11011011BBBCCCCCC0QQQQQ
RCMP<.cc>	b,u6	00100bbb11011011BBBuuuuuu1QQQQQ

RCMPL

Function

Reverse Comparison Long

Extension Group

Baseline

Operation

if (cc) c – b;

Instruction Format

op b,c

Syntax Example

RCMPL<.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 64-bit source operand 1 (B) from 64-bit source operand 2 (C) and subsequently updating the flags.

There is no destination register. Therefore, the result of the subtract is discarded.

RCMPL always sets the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                        /* RCMPL */
 alu = src2 - src1
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[63]
 C_flag = Carry()
 V_flag = Overflow()
```

Assembly Code Example

```
RCMPL r1,r2    ; Subtract r1 from r2 and set the flags on the result
```

Syntax and Encoding

RCMPL always set the flags even though there is no associated flag setting suffix. For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
RCMPL	b,c	01011bbb00011011BBBCCCCCCRRRRRR
RCMPL	b,u6	01011bbb01011011BBBuuuuuuRRRRRR
RCMPL	b,s12	01011bbb10011011BBBssssssSSSSSS
RCMPL<.cc>	b,c	01011bbb11011011BBBCCCCCC0QQQQQ
RCMPL<.cc>	b,u6	01011bbb11011011BBBuuuuuu1QQQQQ

REM

Function

2's complement integer Remainder.

Extension Group

Baseline

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, refer to Appendix A, "Implementation-dependent Behavior".

Pseudo Code

```
if (cc == true) {                                                    /* REM */
   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 Table 8-7 and Table 8-8.

		Instruction Code
REM<.f>	a,b,c	00101bbb00001000FBBBCCCCCCAAAAAA
REM<.f>	a,b,u6	00101bbb01001000FBBBuuuuuuAAAAAA
REM<.f>	b,b,s12	00101bbb10001000FBBBssssssSSSSSS
REM<.cc><.f>	b,b,c	00101bbb11001000FBBBCCCCCC0QQQQQ
REM<.cc><.f>	b,b,u6	00101bbb11001000FBBBuuuuuu1QQQQQ

REML

Function

2's complement 64-bit integer Remainder

Extension Group

Baseline

Operation

if (cc) then a = b % c;

Instruction Format

op a, b, c

Syntax Example

REML <.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 64-bit integer division of 64-bit source operand (b) by 64-bit 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 64 bits. This overflow only occurs in one specific case, which is the division of the largest representable negative value (0x8000000000000000) 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 64 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, refer to Appendix A, "Implementation-dependent Behavior".

Pseudo Code

```
if (cc == true) {                                                    /* REML */
   if ((src2 != 0) && (    (src1 != 0x8000000000000000
                        || (src2 != 0xffffffffffffffff))) {
        dest = src1 % src2;
        if (F == 1)  {
            Z = (dest == 0) ? 1 : 0;
            N = dest[63];
            V = 0;
        }
   } else {
      if ((src2 == 0) && (STATUS32.DZ == 1))
         RaiseException (EV_DivZero);
      else {
         /* optional implementation-dependent result assignment */
         if (F == 1){
             V = 1;
         }
      }
   }
}
```

Assembly Code Example

```
REML r1,r2,r3                              ; r1 is assigned r2 % r3
```

Syntax and Encoding

		Instruction Code
REML<.f>	a,b,c	01011bbb00101000FBBBCCCCCCAAAAAA
REML<.f>	a,b,u6	01011bbb01101000FBBBuuuuuuAAAAAA
REML<.f>	b,b,s12	01011bbb10101000FBBBssssssSSSSSS
REML<.cc><.f>	b,b,c	01011bbb11101000FBBBCCCCCC0QQQQQ
REML<.cc><.f>	b,b,u6	01011bbb11101000FBBBuuuuuu1QQQQQ

REMU

Function

Unsigned integer Remainder.

Extension Group

Baseline

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, refer to Appendix A, "Implementation-dependent Behavior".

If the flag-update bit is set, an unsigned REMU operation always clears the N-flag.

Pseudo Code

```
if (cc == true) {                                                    /* REMU */
   if (src2 != 0) {
        dest = src1 % src2;
        if (F == 1)  {
            Z = (dest == 0) ? 1 : 0;
            N = 0;
            V = 0;
        }
   } else {
      if (STATUS32.DZ == 1)
         RaiseException (EV_DivZero);
      else {
         /* optional implementation-dependent assignment to dest */
         if (F == 1){
             V = 1;
         }
      }
   }
}
```

Assembly Code Example

```
REMU r1,r2,r3                        ; r1 is assigned r2 % r3
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
REMU<.f>	a,b,c	00101bbb00001001FBBBCCCCCCAAAAAA
REMU<.f>	a,b,u6	00101bbb01001001FBBBuuuuuuAAAAAA
REMU<.f>	b,b,s12	00101bbb10001001FBBBssssssSSSSSS
REMU<.cc><.f>	b,b,c	00101bbb11001001FBBBCCCCCC0QQQQQ
REMU<.cc><.f>	b,b,u6	00101bbb11001001FBBBuuuuuu1QQQQQ

REMUL

Function

Unsigned 64-bit integer Remainder

Extension Group

Baseline

Operation

if (cc) then a = b % c;

Instruction Format

op a, b, c

Syntax Example

REMUL <.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 64-bit divisor (c) is non-zero, the destination register is assigned the remainder obtained by unsigned integer division of 64-bit source operand (b) by 64-bit 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, refer to Appendix A, "Implementation-dependent Behavior".

If the flag-update bit is set, an unsigned REMUL operation always clears the N-flag.

Pseudo Code

```
if (cc == true) {                                                    /* REMUL */
   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 result assignment */
         if (F == 1){
             V = 1;
         }
      }
   }
}
```

Assembly Code Example

```
REMUL r1,r2,r3                          ; r1 is assigned r2 % r3
```

Syntax and Encoding

		Instruction Code
REMUL<.f>	a,b,c	01011bbb00101001FBBBCCCCCCAAAAAA
REMUL<.f>	a,b,u6	01011bbb01101001FBBBuuuuuuAAAAAA
REMUL<.f>	b,b,s12	01011bbb10101001FBBBssssssSSSSSS
REMUL<.cc><.f>	b,b,c	01011bbb11101001FBBBCCCCCC0QQQQQ
REMUL<.cc><.f>	b,b,u6	01011bbb11101001FBBBuuuuuu1QQQQQ

RLC

Function

Rotate Left Through Carry

Extension Group

Baseline

Operation

b=c<<1; b=b|C; C=c>>31;

Note: In this instruction, C represents the carry bit.

Instruction Format

op b,c

Syntax Example

RLC<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

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 Table 8-7 and Table 8-8.

		Instruction Code
RLC<.f>	b,c	00100bbb00101111FBBBCCCCCC001011
RLC<.f>	b,u6	00100bbb01101111FBBBuuuuuu001011

ROL

Function

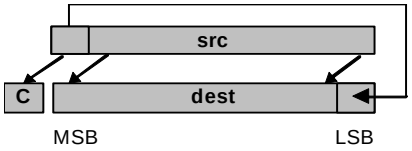
Rotate Left

Extension Group

Baseline

Operation

b = (c << 1) | (c >> 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 Table 8-7 and Table 8-8.

ROL<.f>	b,c	00100bbb00101111FBBBCCCCCC001101
ROL<.f>	b,u6	00100bbb01101111FBBBuuuuuu001101

ROL8

Function

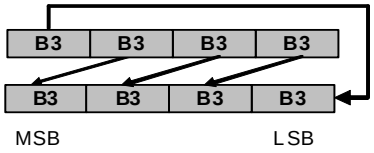
Rotate Left 8

Extension Group

Baseline

Operation

b = (c << 8) | (c >> 24);



Instruction Format

op b,c

Syntax Example

ROL8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Rotate the source operand 8 places to the left.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (src << 8) | (src >> 24)              /* ROR8 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
ROL8 r1,r2   ;  Rotate r2 8 places to the left, placing
             ;  result in r1.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
ROL8<.f>   b,c      00101bbb00101111FBBBCCCCCC010000
ROL8<.f>   b,u6     00101bbb01101111FBBBuuuuuu010000
```

ROR

Function

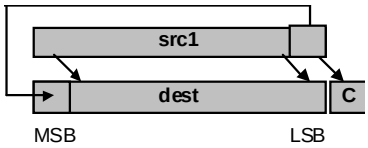
Rotate Right

Extension Group

Baseline

Operation

C = c & 1; b = c>>1 | c<<31;



Instruction Format

op b,c

Syntax Example

ROR<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Rotate the source operand (c) right by one and place the result in the destination register (b).

The least-significant bit of the source operand is copied to the carry flag.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 1                        /* ROR */
dest[31] = src[0]
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = src[0]
```

Assembly Code Example

```
ROR r1,r2    ; Rotate right contents of r2 by one bit and write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
ROR<.f>	b,c	00100bbb00101111FBBBCCCCCC000011
ROR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000011

ROR multiple

Function

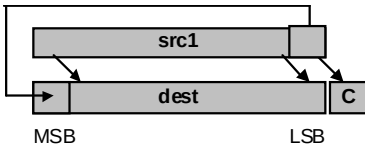
Multiple Rotate Right

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

		Instruction Code
ROR<.f>	a,b,c	00101bbb00000011FBBBCCCCCCAAAAAA
ROR<.f>	a,b,u6	00101bbb01000011FBBBuuuuuuAAAAAA
ROR<.f>	b,b,s12	00101bbb10000011FBBBssssssSSSSSS
ROR<.cc><.f>	b,b,c	00101bbb11000011FBBBCCCCCC0QQQQQ
ROR<.cc><.f>	b,b,u6	00101bbb11000011FBBBuuuuuu1QQQQQ

ROR8

Function

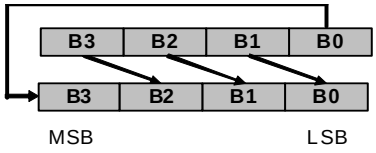
Rotate right 8

Extension Group

Baseline

Operation

b = (c >> 8) | (c << 24);



Instruction Format

op b,c

Syntax Example

ROR8<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Rotate the source operand 8 places to the right.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = (src >> 8) | (src << 24)              /* ROR8 */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
ROR8 r1,r2   ;  Rotate r2 8 places to the right, placing result in r1.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
ROR8<.f>   b,c      00101bbb00101111FBBBCCCCCC010001
ROR8<.f>   b,u6     00101bbb01101111FBBBuuuuuu010001
```

RRC

Function

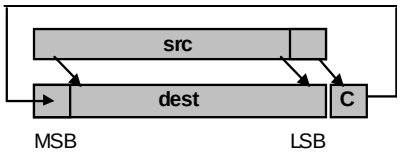
Rotate Right through Carry

Extension Group

Baseline

Operation

b=c>>1 | C << 31; C=c& 1;



Instruction Format

op b,c

Syntax Example

RRC<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Description

Rotate the source operand (c) right by one and place the result in the destination register (a).

The carry flag is shifted into the most-significant bit of the result, and the least-significant bit of the source is placed in the carry flag.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = src >> 1                        /* RRC */
dest[31] = C_flag
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = src[0]
```

Assembly Code Example

```
RRC r1,r2    ; Rotate right through carry contents of r2 by one bit and
             ;write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
RRC<.f>	b,c	00100bbb00101111FBBBCCCCCC000100
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 Table 8-7 and Table 8-8.

		Instruction Code
RSUB<.f>	a,b,c	00100bbb00001110FBBBCCCCCCAAAAAA
RSUB<.f>	a,b,u6	00100bbb01001110FBBBuuuuuuAAAAAA
RSUB<.f>	b,b,s12	00100bbb10001110FBBBssssssSSSSSS
RSUB<.cc><.f>	b,b,c	00100bbb11001110FBBBCCCCCC0QQQQQ
RSUB<.cc><.f>	b,b,u6	00100bbb11001110FBBBuuuuuu1QQQQQ

RSUBL

Function

Reverse Subtract Long

Extension Group

Baseline

Operation

if (cc) a= c-b;

Instruction Format

op a, b, c

Syntax Example

RSUBL<.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 64-bit source operand 1 (b) from 64-bit source operand 2 (c) and place the 64-bit 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                                  /* RSUBL */
 dest = src2 - src1
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
RSUBL r1,r2,r3   ; Subtract contents of r2 from r3 and write result into r1
```

Syntax and Encoding

		Instruction Code
RSUBL<.f>	a,b,c	01011bbb00001110FBBBCCCCCCAAAAAA
RSUBL<.f>	a,b,u6	01011bbb01001110FBBBuuuuuuAAAAAA
RSUBL<.f>	b,b,s12	01011bbb10001110FBBBssssssSSSSSS
RSUBL<.cc><.f>	b,b,c	01011bbb11001110FBBBCCCCCC0QQQQQ
RSUBL<.cc><.f>	b,b,u6	01011bbb11001110FBBBuuuuuu1QQQQQ

RTIE

Function

Return from Interrupt/Exception

Extension Group

Baseline

Operation

```
if ((HAS_INTERRUPTS == 0) || (STATUS32.AE) || (AUX_IRQ_ACT[15:0] == 0))
     ReturnFromException()
else
     if ((AUX_IRQ_ACT[0]) && (FIRQ_OPTION))
        if (IRQ_PRIORITY_PENDING[0])
            ServiceAnotherFastInterrupt()
        else
            ReturnFromFastInterrupt()
     else
        if (PendingNewIntrPreemptsReturnLevel)
            ServiceAnotherInterrupt()
        else
            ReturnFromInterrupt()
```

Note For information about the conditions and function used in this section, see the sections: Fast Interrupt Entry, Returning from Regular Interrupts, Exception Entry, 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)                    //see example 4.5
    else
       ReturnFromInterrupt(PI, W)                         //see example 4.7

ReturnFromException()
  Next_PC  = ERET
  STATUS32 = ERSTATUS
  BTA      = ERBTA
  if (ERSTATUS.U)                                        //check for User mode
      AEX R28,[AUX_USER_SP]                              //restore User SP
  Jump(Next_PC)
```

Assembly Code Example

```
RTIE                                ; Return from interrupt/exception
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

	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 Table 8-7 and Table 8-8.

		Instruction Code
SBC<.f>	a,b,c	00100bbb00000011FBBBCCCCCCAAAAAA
SBC<.f>	a,b,u6	00100bbb01000011FBBBuuuuuuAAAAAA
SBC<.f>	b,b,s12	00100bbb10000011FBBBssssssSSSSSS
SBC<.cc><.f>	b,b,c	00100bbb11000011FBBBCCCCCC0QQQQQ
SBC<.cc><.f>	b,b,u6	00100bbb11000011FBBBuuuuuu1QQQQQ

SBCL

Function

Subtract Long with Carry

Extension Group

Baseline

Operation

if (cc) a = (b – c) - C;

Instruction Format

op a, b, c

Syntax Example

SBCL<.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 64-bit source operand 2 (c) from 64-bit 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                              /* SBCL */
 dest = (src1 - src2) - C_flag
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
SBCL r1,r2,r3   ; Subtract with carry contents of r3 from r2 and write
                ; result into r1
```

Syntax and Encoding

		Instruction Code
SBCL<.f>	a,b,c	01011bbb00000011FBBBCCCCCCAAAAAA
SBCL<.f>	a,b,u6	01011bbb01000011FBBBuuuuuuAAAAAA
SBCL<.f>	b,b,s12	01011bbb10000011FBBBssssssSSSSSS
SBCL<.cc><.f>	b,b,c	01011bbb11000011FBBBCCCCCC0QQQQQ
SBCL<.cc><.f>	b,b,u6	01011bbb11000011FBBBuuuuuu1QQQQQ

SCOND

Function

Store Conditional

Extension Group

Baseline

Operation

_scond (b,c);

Instruction Format

op b,c

Syntax Example

SCOND<.di> b, [c]

STATUS32 Flags Affected

Z	•	= Set to the value of Lock Flag immediately before the SCOND takes place
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

If the Lock Flag register (LF) is set to 1, the 32-bit value from the b source operand register is written to the memory address given by c source operand register. If LF is set to 0, no memory write takes place.

When this instruction completes, whether it writes to memory or not, a copy of the LF register is assigned to the Z status flag and the LF register is cleared.

This instruction always performs the test of LF and the conditional write to memory as an atomic operation. If the operand location has been written, between the point at which LF is set to 1 by LLOCK, and the execution of the following SCOND, the SCOND does not write to memory and clears the Z flag. As a result, if two processors use SCOND instructions to modify the same address, and both have their respective LF registers set to 1, only one performs the store operation. The value returned to each respective processor's Z flag indicates which processor succeeded in writing, and the value 1 is returned to only one such contending processor.

Each processor is expected to test the Z status result, returned by an SCOND instruction, to see if the LLOCK/SCOND pair succeeded. Software typically repeats the LLOCK/SCOND sequence until success is detected. The success or failure of the SCOND instruction is returned to the processor via the Zero

flag (STATUS32[Z]), which is set to 1 if the SCOND instruction succeeded in writing to memory, and is cleared if the write to memory did not take place.

The `SCOND.DI` instruction operates directly on external memory, bypassing any data cache that may be present in the path from the processor to the physical memory system. This form must be used in implementations of the ARCv3 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 ARCv3 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 Table 8-7 and Table 8-8.

```
SCOND<.di>   b,[c]      00100bbb00101111DBBBCCCCCC010001

SCOND<.di>   b,[u6]     00100bbb01101111DBBBuuuuuu010001
```

SCONDL

Function

Store conditional on 64-bit data

Extension Group

Baseline

Operation

_scondl (b,c);

Instruction Format

op b,c

Syntax Example

SCONDL<.aq> 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 SCONDL instruction is similar to the SCOND instruction except that SCONDL operates on 64-bit data objects and (if successful) it stores a pair of 32-bit core registers to the destination memory address. SCONDL, except that it is available only in ARC64 and it stores a 64-bit core register to the destination memory address (when successful).

The effective address of SCONDL 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

```
SCONDL r0,[r2]     ; Store 64-bit register r0 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 Table 8-7 and Table 8-8.

```
SCONDL          b,[c]      01011bbb00101111 0BBBCCCCCC010001

SCONDL          b,[c]      01011bbb00101111 1BBBCCCCCC010001
```

SETcc

Function

Compute Boolean relation between operands

Extension Group

Baseline

Operation

dest ← Relation(src1, 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 Table 8-7 and Table 8-8.

		Instruction Code
SETcc<.f>	a,b,c	00100bbb00iiiiiiFBBBCCCCCCAAAAAA
SETcc<.f>	a,b,u6	00100bbb01iiiiiiFBBBuuuuuuAAAAAA
SETcc<.f>	b,b,s12	00100bbb10iiiiiiFBBBssssssSSSSSS
SETcc<.cc><.f>	b,b,c	00100bbb11iiiiiiFBBBCCCCCC0QQQQQ
SETcc<.cc><.f>	b,b,u6	00100bbb11iiiiiiFBBBuuuuuu1QQQQQ

SETcc encodings

Sub-opcode	iiiiii	Instruction mnemonic	Description	Operand type
0x38	111000	SETEQ	set if equal	Signed or unsigned
0x39	111001	SETNE	set if not equal	Signed or unsigned
0x3a	111010	SETLT	set if less than	Signed
0x3b	111011	SETGE	set if greater or equal	Signed
0x3c	111100	SETLO	set if lower than	Unsigned
0x3d	111101	SETHS	set if higher or same	Unsigned
0x3e	111110	SETLE	set if less than or equal	Signed
0x3f	111111	SETGT	set if greater than	Signed

SETccL

Function

Compute Boolean relation between 64-bit operands

Extension Group

Baseline

Operation

dest ← Relation(src1, src2)

Instruction Format

op a, b, c

Syntax Example

SETEQL<.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 64-bit 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                                    /* SETccL */
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

```
SETLTL r1,r2,r3    ; Set r1 to 1 if r2 < r3 otherwise set r1 to 0
```

Syntax and Encoding

		Instruction Code
SETccL<.f>	a,b,c	01011bbb00iiiiiiFBBBCCCCCCAAAAAA
SETccL<.f>	a,b,u6	01011bbb01iiiiiiFBBBuuuuuuAAAAAA
SETccL<.f>	b,b,s12	01011bbb10iiiiiiFBBBssssssSSSSSS
SETccL<.cc><.f>	b,b,c	01011bbb11iiiiiiFBBBCCCCCC0QQQQQ
SETccL<.cc><.f>	b,b,u6	01011bbb11iiiiiiFBBBuuuuuu1QQQQQ

SETccL encodings

Sub-opcode	iiiiii	Instruction mnemonic	Description	Operand type
0x38	111000	SETEQL	set if equal	Signed or unsigned
0x39	111001	SETNEL	set if not equal	Signed or unsigned
0x3a	111010	SETLTL	set if less than	Signed
0x3b	111011	SETGEL	set if greater or equal	Signed
0x3c	111100	SETLOL	set if lower than	Unsigned
0x3d	111101	SETHSL	set if higher or same	Unsigned
0x3e	111110	SETLEL	set if less than or equal	Signed
0x3f	111111	SETGTL	set if greater than	Signed

SETI

Function

SETI enables interrupts, and if a source operand is mentioned, sets the interrupt threshold in the STATUS32 register.

Extension Group

HAS_INTERRUPTS == 1

Operation

```
{
if (c[5] == 1)
   STATUS32.E ←[3:0]
   STATUS32.IE ←c[4]
   else
     STATUS32.IE ← 1
     if (c[4] == 1)
        STATUS32.E ←c[3:0]
}
```

Instruction Format

op c

Syntax Example

SETI c

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

SETI takes a single u6 or C source register operand that indicates whether the interrupt priority level is set when enabling interrupts, and if so to what level.

When an ARCv3-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

```
{                                                                    /* SETI
if ( src2[5] == 1 )                                                  */
         STATUS32.E <-src2[3:0]
         STATUS32.IE <-src2[4]
      else
         STATUS32.IE <- 1
         if (src2[4] == 1)
          STATUS32.E <- src2[3:0]
}
```

Assembly Code Example

```
SETI R0    ;enable interrupts and restore the preempting interrupt
           ;priority value.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
SETI	c	00100110001011110000CCCCCC111111
SETI	u6	00100110011011110000uuuuuu111111

SETI takes a 6-bit operand. Bit 5 indicates whether interrupt enables are being restored (1) or forced to specific values (0).

When forcing the interrupt enable to 1, the STATUS32.E field can be optionally set according to bits [3:0] of the SETI operand.

Bit5	Bit4	Purpose
0	0	Enable interrupts without changing the enabled level
0	1	Enable interrupts and also set the enabled level
1	x	Restore STATUS32.IE and STATUS32.E using bits 4 and [3:0] respectively

The assembler must encode a SETI without operands as SETI u6, with u6 == 0 in order to provide a concise way of specifying "enable interrupts".

SETI is not a serializing instruction.

SEXB

Function

Sign Extend Byte

Extension Group

Baseline

Operation

b = (c << 24) >> 24;

Instruction Format

op b,c

Syntax Example

SEXB<.f> b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Sign extend the byte contained in the least-significant 8 bits of source operand (c) to a full 32-bit word value and place the result into the destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest[7:0]  = src[7:0]                /* SEXB */
dest[31:8] = src[7]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example

```
SEXB r1,r2   ; Sign extend the bottom 8 bits of r2 and write result to r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
SEXB<.f>	b,c	00100bbb00101111FBBBCCCCCC000101
SEXB<.f>	b,u6	00100bbb01101111FBBBuuuuuu000101
SEXB_S	b,c	01111bbbccc01101

SEXBL

Function

Sign Extend Byte to Long

Extension Group

Baseline

Operation

b = (c << 56) >> 56;

Instruction Format

op b,c

Syntax Example

SEXBL<.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 64-bit long word value and place the result into the 64-bit destination register (b).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest[7:0]  = src[7:0]                    /* SEXBL */
dest[63:8] = src[7]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
SEXBL r1,r2   ; Sign extend the bottom 8 bits of r2 and write result to r1
```

Syntax and Encoding

		Instruction Code
SEXBL<.f>	b,c	01011bbb00101111FBBBCCCCCC000101
SEXBL<.f>	b,u6	01011bbb01101111FBBBuuuuuu000101

SEXHL

Function

Sign Extend Half-word 16-bits to Long

Extension Group

Baseline

Operation

b = (c << 48) >> 48;

Instruction Format

op b,c

Syntax Example

SEXHL<.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 64-bit long word value and place the result into the 64-bit destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest[15:0] = src[15:0]                /* SEXHL */
dest[63:16] = src[15]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
SEXHL r1,r2    ; Sign extend the bottom 16 bits of r2 and write result to r1
```

Syntax and Encoding

		Instruction Code
SEXHL<.f>	b,c	01011bbb00101111FBBBCCCCCC000110
SEXHL<.f>	b,u6	01011bbb01101111FBBBuuuuuu000110

SEXWL

Function

Sign Extend 32-bit Word to Long

Extension Group

Baseline

Operation

b = (c << 32) >> 32;

Instruction Format

op b,c

Syntax Example

SEXWL<.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 32-bit word contained in the least-significant 32 bits of source operand (c) to a full 64-bit long word value and place the result into the 64-bit destination register (a).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest[31:0] = src[31:0]                /* SEXWL */
dest[63:32] = src[15]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
SEXWL r1,r2    ; Sign extend the bottom 32 bits of r2 and write result to r1
```

Syntax and Encoding

		Instruction Code
SEXWL<.f>	b,c	01011bbb00101111FBBBCCCCCC000111
SEXWL<.f>	b,u6	01011bbb01101111FBBBuuuuuu000111

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 11-10
E[3:0]	•	Updated depending on SLEEP operand as explained in Table 11-10
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
ZZ	•	= 1
SM	•	Updated depending on SLEEP operand as explained in Table 11-10

Description

The ARCv3-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 waits for the completion of all previous instructions 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.

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 ARCv3-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 11-1 SLEEP Operand

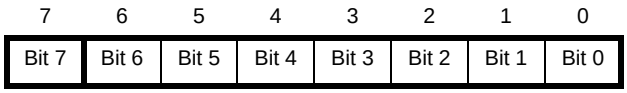


Table 11-10 SLEEP Operand

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 11-5 Sleep placement in code

```
SUB r2, r2, 0x1
ADD r1, r1, 0x2
SLEEP
...
```

Example 11-6 illustrates the use of the SLEEP instruction with each combination of interrupts levels enabled or disabled during sleep.

Example 11-6 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 Table 8-7 and Table 8-8.

		Instruction Code
SLEEP	c	00100001001011111000CCCCCC111111
SLEEP	<u6>	00100001011011111000uuuuuu111111

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 Table 8-7 and Table 8-8.

		Instruction Code
SR	b,[c]	00100bbb00101011RBBBCCCCCCRRRRRR
SR	b,[u6]	00100bbb01101011RBBBuuuuuuRRRRRR
SR	b,[s12]	00100bbb10101011RBBBssssssSSSSSS

SRL

Function

Store Long to Auxiliary Register

Extension Group

Baseline

Operation

_srl(b,c);

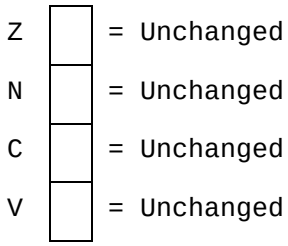
Instruction Format

op b,c

Syntax Example

SRL b,[c]

STATUS32 Flags Affected



Description

Store the 64-bit long word value from source operand 1 (b) into the auxiliary register whose address is obtained from the source operand 2 (c). If c is a register operand then the upper 32 bits are ignored and do not form part of the auxiliary address.

Pseudo Code

```
Aux_reg(src2) = src1    /* SRL */
```

Assembly Code Example

```
SRL r1,[r2]   ; Store 64 bit contents of r1 into Aux. register pointed
              ; to by r2
```

Syntax and Encoding

		Instruction Code
SRL	b,[c]	01011bbb00101011 0BBBCCCCCCRRRRRR
SRL	b,[u6]	01011bbb01101011 0BBBuuuuuuRRRRRR
SRL	b,[s12]	01011bbb10101011 0BBBssssssSSSSSS

STB

Function

Store byte to Memory

Extension Group

Baseline

Operation

addr = b + offset; *addr = c;

Instruction Format

op c, [b, offset]

Syntax Example

STB<.aa><.di> c,[b,s9]

STATUS32 Flags Affected

None

Description

The first source operand specifies the store data and may be a register (C), a long-immediate value (LIMM) or a signed 6-bit constant (w6).

If the store data operand is register or a LIMM, the lower 8 bits of that operand provides 8 bits of store data. If the store data operand is a w6 constant, its value is sign-extended to 8 bits to form the store data value.

The 8-bit store data value is written to memory at the address given by the sum of the second source operand (b) and an offset provided by source operand 3.

Instructions that specify the .DI modifier will bypass any cache present in the system and store directly to memory.

All store-byte instructions may use the .AB, or the .AW addressing modes, but there is no encoding for the .AS mode and it is not supported. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src2
else
  addr = src2 + src3

MemoryWrite(addr, src1[7:0])                /* write 1 byte to memory */

if ((<.aa> == .AB) or (<.aa> == .AS) or (<.aa> == .A))
  src2 = src2 + src3
```

Assembly Code Examples

```
STB r0,[r1,4]; store least-significant byte of r0 to address r1+4
STB 4,[r2,1] ; store the value 4 to the byte address r2+1
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
STB<.aa><.di>	c,[b]	00011bbb00000000 0BBBCCCCCCDaaZZK
STB<.aa><.di>	w6,[b]	00011bbb00000000 0BBBwwwwwwDaaZZK
STB<.aa><.di>	c,[b,s9]	00011bbbssssssss SBBBCCCCCCDaaZZK
STB<.aa><.di>	w6,[b,s9]	00011bbbssssssss SBBBwwwwwwDaaZZK
STB_S	b,[SP,u7]	11000bbb011uuuuu
STB_S	c,[b,u5]	10100bbbcccuuuuu

K ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 01	STB c	STB.DI c	STB.AW c	STB.AW.DI c	STB.AB c	STB.AB.DI c	Illegal	Illegal
1 01	STB w6	STB.DI w6	STB.AW w6	STB.AW.DI w6	STB.AB w6	STB.AB.DI w6	Illegal	Illegal

Note Compact encodings of stack-pointer relative loads have their offset encoded without the least-significant two bits, as all stack addresses are expected to be four-byte aligned. The encoded offset is shifted left by two bit positions creating an offset in which the lower two bits are always 0.

STH

Function

Store half-word to Memory

Extension Group

Baseline

Operation

addr = b + offset; *addr = c;

Instruction Format

op c, [b, offset]

Syntax Example

STB<.aa><.di> c,[b,s9]

STATUS32 Flags Affected

None

Description

The first source operand specifies the store data and may be a register (C), a long-immediate value (LIMM) or a signed 6-bit constant (w6).

If the store data operand is register or a LIMM, the lower 16 bits of that operand provides 16 bits of store data. If the store data operand is a w6 constant, its value is sign-extended to 16 bits to form the store data value.

The 16-bit store data value is written to memory at the address given by the sum of the second source operand (b) and an offset provided by source operand 3. The offset is scaled by a factor of 2 when the .AS addressing mode is specified by an STH instruction.

Instructions that specify the .DI modifier will bypass any cache present in the system and store directly to memory.

All half-word store instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src2
else if (<.aa> == .AS)
  addr = src2 + (src3 << 1)
else
  addr = src2 + src3

MemoryWrite(addr, src1[15:0])            /* write half-word to memory */

if ((<.aa> == .AB) or (<.aa> == .AW))
  src2 = src2 + src3
```

Assembly Code Examples

```
STH r0,[r1,4]    ; store least-significant half-word of r0 to address r1+4
STH 4,[r2,1]     ; store the value 4 to the half-word at address r2+1
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
STB<.aa><.di>	c,[b]	00011bbb00000000 0BBBCCCCCCDaaZZK
STB<.aa><.di>	w6,[b]	00011bbb00000000 0BBBwwwwwwDaaZZK
STB<.aa><.di>	c,[b,s9]	00011bbbssssssss SBBBCCCCCCDaaZZK
STB<.aa><.di>	w6,[b,s9]	00011bbbssssssss SBBBwwwwwwDaaZZK
STB_S	c,[b,u5]	10100bbbcccuuuuu

K ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 10	STH c	STH.DI c	STH.AW c	STH.AW.DI c	STH.AB c	STH.AB.DI c	STH.AS c	STH.AS.DI c
1 10	STH w6	STH.DI w6	STH.AW w6	STH.AW.DI w6	STH.AB w6	STH.AB.DI w6	STH.AS w6	STH.AS.DI w6

Note	STH_S instructions omit the least-significant bit of their encoded offset, as these are assumed to be zero. The encoded constant is shifted left by one bit positions creating an offset in which the least-significant bit is always 0.

Note	For more information on the rules governing non-aligned memory references, see Data Layout in Memory

ST

Function

Store word to Memory

Extension Group

Baseline

Operation

addr = b + offset; *addr = c;

Instruction Format

op c, [b, offset]

Syntax Example

STB<.aa><.di> c,[b,s9]

STATUS32 Flags Affected

None

Description

The first source operand specifies the store data and may be a register (C), a long-immediate value (LIMM) or a signed 6-bit constant (w6).

If the store data operand is a register or a LIMM, that operand provides 32 bits of store data. If the store data operand is a w6 constant, its value is sign-extended to 32 bits to form the store data value.

The 32-bit store data value is written to memory at the address given by the sum of the second source operand (b) and an offset provided by source operand 3. The offset is scaled by a factor of 4 when the .AS addressing mode is specified by an ST instruction.

Instructions that specify the .DI modifier will bypass any cache present in the system and store directly to memory.

All store-word instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src2
else if (<.aa> == .AS)
  addr = src2 + (src3 << 2)
else
  addr = src2 + src3

MemoryWrite(addr, src1[31:0])              /* write word to memory */

if ((<.aa> == .AB) or (<.aa> == .AW))
  src2 = src2 + src3
```

Assembly Code Examples

```
ST r0,[r1,4]     ; store r0 to address r1+4
ST -1,[r2,1]     ; store the value -1 to the word at address r2+1
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
STB<.aa><.di>	c,[b]	00011bbb00000000 0BBBCCCCCCDaaZZK
STB<.aa><.di>	w6,[b]	00011bbb00000000 0BBBwwwwwwDaaZZK
STB<.aa><.di>	c,[b,s9]	00011bbbssssssss SBBBCCCCCCDaaZZK
STB<.aa><.di>	w6,[b,s9]	00011bbbssssssss SBBBwwwwwwDaaZZK
STB_S	c,[b,u7]	10100bbbcccuuuuu
STB_S	b,[SP,u7]	10100bbb010uuuuu
STB_S	R0,[SP,u7]	01010SSSSSS10sss

K ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 00	ST c	ST.DI c	ST.AW c	ST.AW.DI c	ST.AB c	ST.AB.DI c	ST.AS c	ST.AS.DI c
1 00	ST w6	ST.DI w6	ST.AW w6	ST.AW.DI w6	ST.AB w6	ST.AB.DI w6	ST.AS w6	ST.AS.DI w6

Note	ST_S instructions omit the least-significant two bits of their encoded offset, as these are assumed to be zero. The encoded constant is shifted left by two bit positions creating an offset in which the least-significant two bits are always 0.

Note	For more information on the rules governing non-aligned memory references, see Data Layout in Memory

STB

Function

Store byte to Memory

Extension Group

Baseline

Operation

addr = b + offset; *addr = c;

Instruction Format

op c, [b, offset]

Syntax Example

STB<.aa><.di> c,[b,s9]

STATUS32 Flags Affected

None

Description

The first source operand specifies the store data and may be a register (C), a long-immediate value (LIMM) or a signed 6-bit constant (w6).

If the store data operand is register or a LIMM, the lower 8 bits of that operand provides 8 bits of store data. If the store data operand is a w6 constant, its value is sign-extended to 8 bits to form the store data value.

The 8-bit store data value is written to memory at the address given by the sum of the second source operand (b) and an offset provided by source operand 3.

Instructions that specify the .DI modifier will bypass any cache present in the system and store directly to memory.

All store-byte instructions may use the .AB, or the .AW addressing modes, but there is no encoding for the .AS mode and it is not supported. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src2
else
  addr = src2 + src3

MemoryWrite(addr, src1[7:0])                /* write 1 byte to memory */

if ((<.aa> == .AB) or (<.aa> == .AS) or (<.aa> == .A))
  src2 = src2 + src3
```

Assembly Code Examples

```
STB r0,[r1,4]; store least-significant byte of r0 to address r1+4
STB -1,[r2,1]     ; store the value 0xff to the byte address r2+1
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
STB<.aa><.di>	c,[b]	00011bbb00000000 0BBBCCCCCCDaaZZK
STB<.aa><.di>	w6,[b]	00011bbb00000000 0BBBwwwwwwDaaZZK
STB<.aa><.di>	c,[b,s9]	00011bbbssssssss SBBBCCCCCCDaaZZK
STB<.aa><.di>	w6,[b,s9]	00011bbbssssssss SBBBwwwwwwDaaZZK
STB_S	b,[SP,u7]	11000bbb011uuuuu
STB_S	c,[b,u5]	10100bbbcccuuuuu

K ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 01	STB c	STB.DI c	STB.AW c	STB.AW.DI c	STB.AB c	STB.AB.DI c	Illegal	Illegal
1 01	STB w6	STB.DI w6	STB.AW w6	STB.AW.DI w6	STB.AB w6	STB.AB.DI w6	Illegal	Illegal

Note Compact encodings of stack-pointer relative loads have their offset encoded without the least-significant two bits, as all stack addresses are expected to be four-byte aligned. The encoded offset is shifted left by two bit positions creating an offset in which the lower two bits are always 0.

STH

Function

Store half-word to Memory

Extension Group

Baseline

Operation

addr = b + offset; *addr = c;

Instruction Format

op c, [b, offset]

Syntax Example

STB<.aa><.di> c,[b,s9]

STATUS32 Flags Affected

None

Description

The first source operand specifies the store data and may be a register (C), a long-immediate value (LIMM) or a signed 6-bit constant (w6).

If the store data operand is register or a LIMM, the lower 16 bits of that operand provides 16 bits of store data. If the store data operand is a w6 constant, its value is sign-extended to 16 bits to form the store data value.

The 16-bit store data value is written to memory at the address given by the sum of the second source operand (b) and an offset provided by source operand 3. The offset is scaled by a factor of 2 when the .AS addressing mode is specified by an STH instruction.

Instructions that specify the .DI modifier will bypass any cache present in the system and store directly to memory.

All half-word store instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src2
else if (<.aa> == .AS)
  addr = src2 + (src3 << 1)
else
  addr = src2 + src3

MemoryWrite(addr, src1[15:0])            /* write half-word to memory */

if ((<.aa> == .AB) or (<.aa> == .AW))
  src2 = src2 + src3
```

Assembly Code Examples

```
STH    r0,[r1,4] ; store least-significant half-word of r0 to address r1+4
STH.AS -1,[r2,1] ; store the value 0xffff to memory at address r2+2
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
STH<.aa><.di>	c,[b]	00011bbb00000000 0BBBCCCCCCDaaZZK
STH<.aa>	w6,[b]	00011bbb00000000 0BBBwwwwwwDaaZZK
STH<.aa><.di>	c,[b,s9]	00011bbbssssssss SBBBCCCCCCDaaZZK
STH<.aa>	w6,[b,s9]	00011bbbssssssss SBBBwwwwwwDaaZZK
STH_S	c,[b,u6]	10100bbbcccuuuuu

K ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 10	STH c	STH.DI c	STH.AW c	STH.AW.DI c	STH.AB c	STH.AB.DI c	STH.AS c	STH.AS. DI c
1 10	STH w6	STDL	STH.AW w6	STDL	STH.AB w6	STDL	STH.AS w6	STDL

Note	STH_S instructions omit the least-significant bit of their encoded offset, as these are assumed to be zero. The encoded constant is shifted left by one bit positions creating an offset in which the least-significant bit is always 0.

Note	For more information on the rules governing non-aligned memory references, see Data Layout in Memory

ST

Function

Store word to Memory

Extension Group

Baseline

Operation

addr = b + offset; *addr = c;

Instruction Format

op c, [b, offset]

Syntax Example

STB<.aa><.di> c,[b,s9]

STATUS32 Flags Affected

None

Description

The first source operand specifies the store data and may be a register (C), a long-immediate value (LIMM) or a signed 6-bit constant (w6).

If the store data operand is a register or a LIMM, that operand provides 32 bits of store data. If the store data operand is a w6 constant, its value is sign-extended to 32 bits to form the store data value.

The 32-bit store data value is written to memory at the address given by the sum of the second source operand (b) and an offset provided by source operand 3. The offset is scaled by a factor of 4 when the .AS addressing mode is specified by an ST instruction.

Instructions that specify the .DI modifier will bypass any cache present in the system and store directly to memory.

All store-word instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src2
else if (<.aa> == .AS)
  addr = src2 + (src3 << 2)
else
  addr = src2 + src3

MemoryWrite(addr, src1[31:0])              /* write word to memory */

if ((<.aa> == .AB) or (<.aa> == .AW))
  src2 = src2 + src3
```

Assembly Code Examples

```
ST    r0,[r1,4]  ; store r0 to address r1+4
ST.AS -1,[r2,1]  ; store the value 0xffffffff to memory at address r2+4
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
ST<.aa><.di>	c,[b]	00011bbb00000000BBBCCCCCCDaaZZK
ST<.aa><.di>	w6,[b]	00011bbb00000000BBBwwwwwwDaaZZK
ST<.aa><.di>	c,[b,s9]	00011bbbssssssssSBBBCCCCCCDaaZZK
ST<.aa><.di>	w6,[b,s9]	00011bbbssssssssSBBBwwwwwwDaaZZK
ST_S	c,[b,u7]	10100bbbcccuuuuu
ST_S	b,[SP,u7]	10100bbb010uuuuu
ST_S	R0,[SP,u7]	01010SSSSSS10sss

K ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 00	ST c	ST.DI c	ST.AW c	ST.AW.DI c	ST.AB c	ST.AB.DI c	ST.AS c	ST.AS.DI c
1 00	ST w6	ST.DI w6	ST.AW w6	ST.AW.DI w6	ST.AB w6	ST.AB.DI w6	ST.AS w6	ST.AS.DI w6

Note	ST_S instructions omit the least-significant two bits of their encoded offset, as these are assumed to be zero. The encoded constant is shifted left by two bit positions creating an offset in which the least-significant two bits are always 0.

Note	For more information on the rules governing non-aligned memory references, see Data Layout in Memory

STL

Function

Store long-word to Memory

Extension Group

Baseline

Operation

addr = b + offset; *addr = c;

Instruction Format

op c, [b, offset]

Syntax Example

STB<.aa><.di> c,[b,s9]

STATUS32 Flags Affected

None

Description

The first source operand specifies the store data and may be a register (C), a signed or unsigned long-immediate value (LIMM) or a signed 6-bit constant (w6).

If the store data operand is register it provides a 64-bit of store data value. If the store data operand is a signed LIMM or a w6 constant, its value is sign-extended to 64 bits to form the store data value. If the store data operand is an unsigned LIMM, its value is zero-extended to 64 bits to form the store data value.

The 64-bit store data value is written to memory at the address given by the sum of the second source operand (b) and an offset provided by source operand 3. The offset is scaled by a factor of 8 when the .AS addressing mode is specified by an STL instruction.

All store-word instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src2
else if (<.aa> == .AS)
  addr = src2 + (src3 << 3)
else
  addr = src2 + src3

MemoryWrite(addr, src1[63:0])          /* write double-word to memory */

if ((<.aa> == .AB) or (<.aa> == .A))
  src2 = src2 + src3
```

Assembly Code Examples

```
STL r0,[r1,4]    ; store r0 to address r1+4
STL.AS -1,[r2,1] ; store 0xffffffffffffffff to memory at address r2+8
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
STL<.aa>	c,[b]	00011bbb00000000 0BBBCCCCCCDaaZZK
STL<.aa>	w6,[b]	00011bbb00000000 0BBBwwwwwwDaaZZK
STL<.aa>	c,[b,s9]	00011bbbssssssss SBBBCCCCCCDaaZZK
STL<.aa>	w6,[b,s9]	00011bbbssssssss SBBBwwwwwwDaaZZK

K ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
1 11	STL c	STL w6	STL.AW w6	STL.AW w6	STL.AB c	STL.AB w6	STL.AS	STL.AS w6

Note For more information on the rules governing non-aligned memory references, see Data Layout in Memory

STDL

Function

Store double long-word to Memory

Extension Group

M128_OPTION

Operation

addr = b + offset; *addr = c;

Instruction Format

op c, [b, offset]

Syntax Example

STDL<.aa> C,[b,s9]

STATUS32 Flags Affected

None

Description

The first source operand specifies the store data and may be an even-numbered pair of registers (C), a signed or unsigned long-immediate value (LIMM) or a signed 6-bit constant (w6).

If the store data operand is register pair, register C supplies the lower eight bytes of store data and register C+1 supplies the upper eight bytes of store data. An Illegal Instruction exception is raised if C is an odd-numbered register.

If the store data operand is a signed LIMM or a w6 constant, its value is sign-extended to 64 bits to form a scalar store data element. If the store data operand is an unsigned LIMM, its value is zero-extended to 64 bits to form a scalar store data element. The scalar store data element is then duplicated to form a 128-bit store data value.

The 128-bit store data value is written to memory at the address given by the sum of the second source operand (b) and an offset provided by source operand 3. The offset is scaled by a factor of 8 when the .AS addressing mode is specified by an STDL instruction.

All double-long-word store instructions may use the .AB, .AW and .AS addressing modes. See Semantics of Load / Store Address Write-back Modes for details of all addressing modes supported by the instruction set.

Pseudo Code

```
if (<.aa> == .AB)
  addr = src2
else if (<.aa> == .AS)
  addr = src2 + (src3 << 2)
else
  addr = src2 + src3

MemoryWrite(addr, src1[63:0])          /* write double-word to memory */

if ((<.aa> == .AB) or (<.aa> == .A))
  src2 = src2 + src3
```

Assembly Code Examples

```
STDL r0,[r1,4]              ; store r0 to memory at address r1+4,
                            ; and store r1 to memory address r1+12
STDL 0x87654321@s32,[r0,8] ; store two copies of 0xffffffff87654321, one
                            ; to memory at address r0+8, and one to memory
                            ; at address r0+16
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
STDL<.aa>	c,[b]	00011bbb00000000BBBCCCCCCDaaZZK
STDL<.aa>	w6,[b]	00011bbb00000000BBBwwwwwwDaaZZK
STDL<.aa>	c,[b,s9]	00011bbbssssssssSBBBCCCCCCDaaZZK
STDL<.aa>	w6,[b,s9]	00011bbbssssssssSBBBwwwwwwDaaZZK

K ZZ encodings	aa D encodings							
	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1
0 11	Illegal	STDL c	Illegal	STDL.AW c	Illegal	STDL.AB c	Illegal	STDL.AS c
1 10	STH	STDL w6	STH	STDL.AW w6	STH	STDL.AB w6	STH	STDL.AS w6

Note For more information on the rules governing non-aligned memory references, see Data Layout in Memory

SUB

Function

Subtract

Extension Group

Baseline

Operation

if (cc) a = b – c

Instruction Format

op a, b, c

Syntax Example

SUB<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated

Description

Subtract source operand 2 (c) from source operand 1 (b) and place the result in the destination register a.

SUB_S.NE is executed when the Z flag is equal to zero.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

For the 16-bit encoded instructions that work on the stack pointer (SP), the offset is aligned to 64-bit.

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 Table 8-7 and Table 8-8.

		Instruction Code
SUB<.f>	a,b,c	00100bbb00000010FBBBCCCCCCAAAAAA
SUB<.f>	a,b,u6	00100bbb01000010FBBBuuuuuuAAAAAA
SUB<.f>	b,b,s12	00100bbb10000010FBBBssssssSSSSSS
SUB<.cc><.f>	b,b,c	00100bbb11000010FBBBCCCCCC0QQQQQ
SUB<.cc><.f>	b,b,u6	00100bbb11000010FBBBuuuuuu1QQQQQ
SUB_S.NE	b,b,b	01111bbb11000000
SUB_S	b,b,c	01111bbbccc00010
SUB_S	b,b,u5	10111bbb011uuuuu
SUB_S	SP,SP,u7	Synonymous with SUBL_S SP, SP, u9

Encodings supported by the CODE DENSITY options

SUB_S	a,b,c	01001bbbccc10aaa

SUBL

Function

Subtract Long

Extension Group

Baseline

Operation

if (cc) a = b – c

Instruction Format

op a, b, c

Syntax Example

SUBL<.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 64-bit source operand 2 (c) from 64-bit source operand 1 (b) and place the 64-bit 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                                /* SUBL */
 dest = src1 - src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
SUBL r1,r2,r3    ; Subtract contents of r3 from r2 and write result into r1
```

Syntax and Encoding

		Instruction Code
SUBL<.f>	a,b,c	01011bbb00000010FBBBCCCCCCAAAAAA
SUBL<.f>	a,b,u6	01011bbb01000010FBBBuuuuuuAAAAAA
SUBL<.f>	b,b,s12	01011bbb10000010FBBBssssssSSSSSS
SUBL<.cc><.f>	b,b,c	01011bbb11000010FBBBCCCCCC0QQQQQ
SUBL<.cc><.f>	b,b,u6	01011bbb11000010FBBBuuuuuu1QQQQQ
SUBL_S<.cc><.f>	b,b,c	01111bbbccc00011
SUBL_S<.cc><.f>	SP,SP,u9	11000UU1101uuuuu

SUB1

Function

Subtract with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b – (c << 1)

Instruction Format

op a, b, c

Syntax Example

SUB1<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left shifted by 1) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                        /* SUB1 */
 dest = src1 - (src2 << 1)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
SUB1 r1,r2,r3   ; Subtract contents of r3 left-shifted one bit from r2
                ;and write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
SUB1<.f>	a,b,c	00100bbb00010111FBBBCCCCCCAAAAAA
SUB1<.f>	a,b,u6	00100bbb01010111FBBBuuuuuuAAAAAA
SUB1<.f>	b,b,s12	00100bbb10010111FBBBssssssSSSSSS
SUB1<.cc><.f>	b,b,c	00100bbb11010111FBBBCCCCCC0QQQQQ
SUB1<.cc><.f>	b,b,u6	00100bbb11010111FBBBuuuuuu1QQQQQ

SUB1L

Function

Subtract Long with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b – (c << 1)

Instruction Format

op a, b, c

Syntax Example

SUB1L<.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 64-bit source operand 2 (c) (c left shifted by 1) from 64-bit source operand 1 (b), and place the result in the 64-bit 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                        /* SUB1L */
 dest = src1 - (src2 << 1)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
SUB1L r1,r2,r3   ; Subtract contents of r3 left-shifted one bit from r2
                 ; and write result into r1
```

Syntax and Encoding

		Instruction Code
SUB1L<.f>	a,b,c	01011bbb00010111FBBBCCCCCCAAAAAA
SUB1L<.f>	a,b,u6	01011bbb01010111FBBBuuuuuuAAAAAA
SUB1L<.f>	b,b,s12	01011bbb10010111FBBBssssssSSSSSS
SUB1L<.cc><.f>	b,b,c	01011bbb11010111FBBBCCCCCC0QQQQQ
SUB1L<.cc><.f>	b,b,u6	01011bbb11010111FBBBuuuuuu1QQQQQ

SUB2

Function

Subtract with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b – (c << 2)

Instruction Format

op a, b, c

Syntax Example

SUB2<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left-shifted by 2) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                        /* SUB2 */
 dest = src1 - (src2 << 2)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
SUB2 r1,r2,r3    ; Subtract contents of r3 leftshifted two
                 ; bits from r2 and write resultinto r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
SUB2<.f>	a,b,c	00100bbb00011000FBBBCCCCCCAAAAAA
SUB2<.f>	a,b,u6	00100bbb01011000FBBBuuuuuuAAAAAA
SUB2<.f>	b,b,s12	00100bbb10011000FBBBssssssSSSSSS
SUB2<.cc><.f>	b,b,c	00100bbb11011000FBBBCCCCCC0QQQQQ
SUB2<.cc><.f>	b,b,u6	00100bbb11011000FBBBuuuuuu1QQQQQ

SUB2L

Function

Subtract Long with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b – (c << 2)

Instruction Format

op a, b, c

Syntax Example

SUB2L<.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 64-bit source operand 2 (c) (c left-shifted by 2) from 64-bit source operand 1 (b), and place the result in the 64-bit 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                        /* SUB2L */
 dest = src1 - (src2 << 2)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
SUB2L r1,r2,r3   ; Subtract contents of r3 leftshifted two
                 ; bits from r2 and write resultinto r1
```

Syntax and Encoding

		Instruction Code
SUB2L<.f>	a,b,c	01011bbb00011000FBBBCCCCCCAAAAAA
SUB2L<.f>	a,b,u6	01011bbb01011000FBBBuuuuuuAAAAAA
SUB2L<.f>	b,b,s12	01011bbb10011000FBBBssssssSSSSSS
SUB2L<.cc><.f>	b,b,c	01011bbb11011000FBBBCCCCCC0QQQQQ
SUB2L<.cc><.f>	b,b,u6	01011bbb11011000FBBBuuuuuu1QQQQQ

SUB3

Function

Subtract with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b – (c << 3)

Instruction Format

op a, b, c

Syntax Example

SUB3<.f> a,b,c

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if an overflow is generated from the SUB part of the instruction

Description

Subtract a scaled version of source operand 2 (c) (c left shifted by 3) from source operand 1 (b), and place the result in the destination register a.

If the carry flag is set upon performing the subtraction, the carry flag must be interpreted as a *borrow*.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
if cc==true then                        /* SUB3 */
 dest = src1 - (src2 << 3)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
SUB3 r1,r2,r3   ; Subtract contents of r3 left shifted three
                ; bits from r2 and write result into r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
SUB3<.f>	a,b,c	00100bbb00011001FBBBCCCCCCAAAAAA
SUB3<.f>	a,b,u6	00100bbb01011001FBBBuuuuuuAAAAAA
SUB3<.f>	b,b,s12	00100bbb10011001FBBBssssssSSSSSS
SUB3<.cc><.f>	b,b,c	00100bbb11011001FBBBCCCCCC0QQQQQ
SUB3<.cc><.f>	b,b,u6	00100bbb11011001FBBBuuuuuu1QQQQQ

SUB3L

Function

Subtract Long with Scaled Source

Extension Group

Baseline

Operation

if (cc) a = b – (c << 3)

Instruction Format

op a, b, c

Syntax Example

SUB3L<.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 64-bit source operand 2 (c) (c left shifted by 3) from 64-bit source operand 1 (b), and place the result in the 64-bit 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                          /* SUB3L */
 dest = src1 - (src2 << 3)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example

```
SUB3L r1,r2,r3    ; Subtract contents of r3 left shifted three
                  ; bits from r2 and write result into r1
```

Syntax and Encoding

		Instruction Code
SUB3L<.f>	a,b,c	01011bbb00011001FBBBCCCCCCAAAAAA
SUB3L<.f>	a,b,u6	01011bbb01011001FBBBuuuuuuAAAAAA
SUB3L<.f>	b,b,s12	01011bbb10011001FBBBssssssSSSSSS
SUB3L<.cc><.f>	b,b,c	01011bbb11011001FBBBCCCCCC0QQQQQ
SUB3L<.cc><.f>	b,b,u6	01011bbb11011001FBBBuuuuuu1QQQQQ

SWAP

Function

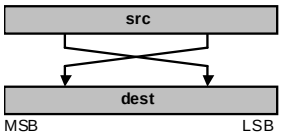
Swap 16-bits half-words

Extension Group

Baseline

Operation

b = c>>16 | (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 Table 8-7 and Table 8-8.

```
SWAP<.f>   b,c      00101bbb00101111FBBBCCCCCC000000
SWAP<.f>   b,u6     00101bbb01101111FBBBuuuuuu000000
```

SWAPL

Function

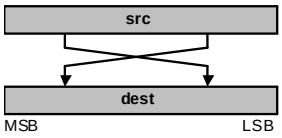
Swap 32-bits words

Extension Group

Baseline

Operation

b = c>>32 | (c & 0xFFFFFFFF)<<32;



Instruction Format

op b,c

Syntax Example

SWAPL<.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 32 bits of the 64-bit operand with the upper 32 bits of the operand and place the result of that swap in the 64-bit destination register.

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
dest = SWAPL(src)                       /* SWAPL */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
SWAPL r1,r2   ; Swap top and bottom 16 bits of r2 write result into r1
```

Syntax and Encoding

SWAPL<.f>	b,c	01011bbb00101111FBBBCCCCCC100000
SWAPL<.f>	b,u6	01011bbb01101111FBBBuuuuuu100000

SWAPE

Function

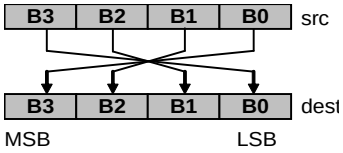
Swap byte ordering

Extension Group

Baseline

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 Table 8-7 and Table 8-8.

SWAPE<.f>	b,c	00101bbb00101111FBBBCCCCCC001001
SWAPE<.f>	b,u6	00101bbb01101111FBBBuuuuuu001001

SWAPEL

Function

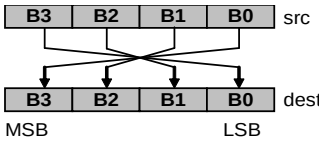
Swap byte ordering long

Extension Group

Baseline

Operation

b = swap order of bytes from c



Instruction Format

op b,c

Syntax Example

SWAPEL<.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 = (SWAPE(src[31:0])<<32)                    /* SWAPEL */
     | SWAPE(src[63:32]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
SWAPEL r1,r2   ; Copy bytes from r2, in reverse order, to r1
```

Syntax and Encoding

SWAPEL<.f>	b,c	01011bbb00101111FBBBCCCCCC101001
SWAPEL<.f>	b,u6	01011bbb01101111FBBBuuuuuu101001

SWI

Function

Software Interrupt or Software Breakpoint

Extension Group

Baseline

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 ERET) is set to the address of the SWI instruction itself. The Exception Fault Address register (see 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 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 Table 8-7 and Table 8-8.

		Instruction Code
SWI		00100010010111011110000RRRRRR111111
SWI_S		0111101011100000
SWI_S	n6*	01111nnnnnn11111

*Where n6 != 111111

SYNC

Function

Synchronize

Extension Group

Baseline

Operation

Wait for all instruction and 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 all previous instructions have committed and have retired their results. The SYNC instruction also waits until all outstanding data memory transactions have completed, and all pending speculative instruction fetches to memory have finished.

Note

The SYNC instruction does not wait on memory operations started by other processors, nor does it wait for the raising of imprecise exceptions from preceding instructions on the same processor.

Use with Interrupts

The SYNC instruction can also be used to ensure that the interrupt request of a memory mapped peripheral has been cleared before an interrupt handler exits.

For example, consider the following scenario:

- A peripheral generates interrupt to the processor by setting a signal to true.

- The control registers for the peripheral are memory mapped
- The processor's interrupt unit is set to 'level sensitive' for this interrupt.
- The interrupt handler must clear the interrupt request signal before exiting

The SYNC instruction is used to ensure that the store to change the peripheral status happens before the interrupt exit.

If the SYNC was not used, the peripheral may still be asserting the interrupt-request signal after the interrupt exit – hence a bogus interrupt is generated.

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
- The DSYNC instruction can also be used more specifically for this purpose.

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          /* pending
           dcache_flush_instruction_retirement_pending)             post-
                                                                    commit
                                                                    result
                                                                    writes */
```

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 Table 8-7 and Table 8-8.

	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 ECR) as the cause parameter along with the cause code for a trap and the trap vector number.

The source value can be used to signal a type of command to any operating system that is running on the processor. Source values 0 to 63 can be used to encode distinct operating system calls.

If implemented in the processor, the Exception Fault Address register (see ECR) is set to point to the address of the trap instruction. The Exception Return Address register (see ERET) is set to the address of the instruction immediately following the trap instruction.

When the exception handler has completed, program execution resumes at the instruction immediately following the trap instruction.

Pseudo Code

```
ERET = NEXTPC                    /* TRAP_S */
ERSTATUS = STATUS32
if STATUS32[DE] == 1 then
  ERBTA = pending PC
ECR = 0x00 : 0x26 : 0x00 : src
EFA = PC
STATUS32[U] = 0
STATUS32[IE] = 0
STATUS32[AE] = 1
PC = INT_VECTOR_BASE + 0x24
```

Assembly Code Example

```
TRAP_S 0                          ; Trap
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

	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 Table 8-7 and Table 8-8.

		Instruction Code
TST	b,c	00100bbb00010111BBBCCCCCCRRRRRR
TST	b,u6	00100bbb01010111BBBuuuuuuRRRRRR
TST	b,s12	00100bbb10010111BBBssssssSSSSSS
TST<.cc>	b,c	00100bbb11010111BBBCCCCCC0QQQQQ
TST<.cc>	b,u6	00100bbb11010111BBBuuuuuu1QQQQQ
TST_S	b,c	01111bbbccc01011

TSTL

Function

Test Long

Extension Group

Baseline

Operation

if (cc) b & c;

Instruction Format

op b,c

Syntax Example

TSTL<.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

Perform bitwise AND 64-bit source operand 1 (b) with 64-bit source operand 2 (c) and subsequently update the flags according to the result.

There is no destination register. Therefore, the result of the AND is discarded.

TSTL alway sets the flags even though there is no associated flag setting suffix.

Pseudo Code

```
if cc==true then                        /* TSTL */
 alu = src1 AND src2
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[63]
```

Assembly Code Example

```
TSTL 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.

		Instruction Code
TSTL	b,c	01011bbb00001011 1BBBCCCCCCRRRRRR
TSTL	b,u6	01011bbb01001011 1BBBuuuuuuRRRRRR
TSTL	b,s12	01011bbb10001011 1BBBssssssSSSSSS
TSTL<.cc>	b,c	01011bbb11001011 1BBBCCCCCC0QQQQQ
TSTL<.cc>	b,u6	01011bbb11001011 1BBBuuuuuu1QQQQQ

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 Table 8-7 and Table 8-8.

	Instruction Code
UNIMP_S	01111001111100000

VADD2

Function

Dual 32-bit addition.

Extension Group

Baseline

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.

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 , 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 Table 8-7 and Table 8-8.

		Instruction Code
VADD2	a,b,c	00101bbb00111100 0BBBCCCCCCAAAAAA
VADD2	a,b,u6	00101bbb01111100 0BBBuuuuuuAAAAAA
VADD2	b,b,s12	00101bbb10111100 0BBBssssssSSSSSS
VADD2<.cc>	b,b,c	00101bbb11111100 0BBBCCCCCC0QQQQQ
VADD2<.cc>	b,b,u6	00101bbb11111100 0BBBuuuuuu1QQQQQ

VADD2H

Function

Dual 16-bit SIMD addition.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
VADD2H	a,b,c	00101bbb00010100 0BBBCCCCCCAAAAAA
VADD2H	a,b,u6	00101bbb01010100 0BBBuuuuuuAAAAAA
VADD2H	b,b,s12	00101bbb10010100 0BBBssssssSSSSSS
VADD2H<.cc>	b,b,c	00101bbb11010100 0BBBCCCCCC0QQQQQ
VADD2H<.cc>	b,b,u6	00101bbb11010100 0BBBuuuuuu1QQQQQ

VADD4H

Function

Quad 16-bit SIMD addition.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
VADD4H	a,b,c	00101bbb00111000 0BBBCCCCCCAAAAAA
VADD4H	a,b,u6	00101bbb01111000 0BBBuuuuuuAAAAAA
VADD4H	b,b,s12	00101bbb10111000 0BBBssssssSSSSSS
VADD4H<.cc>	b,b,c	00101bbb11111000 0BBBCCCCCC0QQQQQ
VADD4H<.cc>	b,b,u6	00101bbb11111000 0BBBuuuuuu1QQQQQ

VADDSUB

Function

Dual 32-bit SIMD add and subtract.

Extension Group

Baseline

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.

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 , 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 Table 8-7 and Table 8-8.

		Instruction Code
VADDSUB	a,b,c	00101bbb00111110 0BBBCCCCCCAAAAAA
VADDSUB	a,b,u6	00101bbb01111110 0BBBuuuuuuAAAAAA
VADDSUB	b,b,s12	00101bbb10111110 0BBBssssssSSSSSS
VADDSUB<.cc>	b,b,c	00101bbb11111110 0BBBCCCCCC0QQQQQ
VADDSUB<.cc>	b,b,u6	00101bbb11111110 0BBBuuuuuu1QQQQQ

VADDSUB2H

Function

Dual 16-bit SIMD add and subtract.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
VADDSUB2H	a,b,c	00101bbb00010110 0BBBCCCCCCAAAAAA
VADDSUB2H	a,b,u6	00101bbb01010110 0BBBuuuuuuAAAAAA
VADDSUB2H	b,b,s12	00101bbb10010110 0BBBssssssSSSSSS
VADDSUB2H<.cc>	b,b,c	00101bbb11010110 0BBBCCCCCC0QQQQQ
VADDSUB2H<.cc>	b,b,u6	00101bbb11010110 0BBBuuuuuu1QQQQQ

VADDSUB4H

Function

Quad 16-bit SIMD add and subtract.

Extension Group

Baseline

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.

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, 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 Table 8-7 and Table 8-8.

		Instruction Code
VADDSUB4H	a,b,c	00101bbb00111010 0BBBCCCCCCAAAAAA
VADDSUB4H	a,b,u6	00101bbb01111010 0BBBuuuuuuAAAAAA
VADDSUB4H	b,b,s12	00101bbb10111010 0BBBssssssSSSSSS
VADDSUB4H<.cc>	b,b,c	00101bbb11111010 0BBBCCCCCC0QQQQQ
VADDSUB4H<.cc>	b,b,u6	00101bbb11111010 0BBBuuuuuu1QQQQQ

VPACK2HL

Function

Compose the destination operand from the lower 16-bits of the source operands.

Extension Group

baseline

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	00101bbb00101001 0BBBCCCCCCAAAAAA
VPACK2HL	a,b,u6	00101bbb01101001 0BBBuuuuuuAAAAAA
VPACK2HL	b,b,s12	00101bbb10101001 0BBBssssssSSSSSS
VPACK2HL<.cc>	b,b,c	00101bbb11101001 0BBBCCCCCC0QQQQQ
VPACK2HL<.cc>	b,b,u6	00101bbb11101001 0BBBuuuuuu1QQQQQ

VPACK2HM

Function

Compose the destination operand from the higher 16-bits of the source operands.

Extension Group

baseline

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 higher16-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	00101bbb00101001 1BBBCCCCCCAAAAAA
VPACK2HM	a,b,u6	00101bbb01101001 1BBBuuuuuuAAAAAA
VPACK2HM	b,b,s12	00101bbb10101001 1BBBssssssSSSSSS
VPACK2HM<.cc>	b,b,c	00101bbb11101001 1BBBCCCCCC0QQQQQ
VPACK2HM<.cc>	b,b,u6	00101bbb11101001 1BBBuuuuuu1QQQQQ

VPACK4HL

Function

Compose the destination result using the even-numbered half-words from the source operands

Extension Group:

Baseline

Operation:

```
a = (c.h2 << 48) | (c.h0 << 32) | (b.h2 << 16) | b.h0
```

Instruction Format

```
op a, b, c
```

Syntax Example

```
VPACK4HL a, b, c
```

STATUS32 Flags Affected:

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Compose the result of the instruction from the even-numbered half-words obtained from the two source operands. This instruction does not modify any flags.

Pseudo Code

```
a = (c.h2 << 48) | (c.h0 << 32) | (b.h2 << 16) | b.h0
```

Assembly Code Example

```
VPACK4HL r0,r1,r2     ;; pack even half-words of r1 and r2 into 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 Table 8-7 and Table 8-8.

		Instruction Code
VPACK4HL	a,b,c	01011bbb00110000 1BBBCCCCCCAAAAAA
VPACK4HL	a,b,u6	01011bbb01110000 1BBBuuuuuuAAAAAA
VPACK4HL	b,b,s12	01011bbb10110000 1BBBssssssSSSSSS
VPACK4HL<.cc>	b,b,c	01011bbb11110000 1BBBCCCCCC0QQQQQ
VPACK4HL<.cc>	b,b,u6	01011bbb11110000 1BBBuuuuuu1QQQQQ

VPACK4HM

Function

Compose the destination result using the odd-numbered half-words from the source operands

Extension Group:

Baseline

Operation:

```
a = (c.h3 << 48) | (c.h1 << 32) | (b.h3 << 16) | b.h1
```

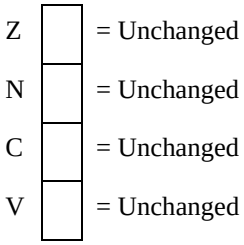
Instruction Format

```
op a, b, c
```

Syntax Example

```
VPACK4HM a, b, c
```

STATUS32 Flags Affected



Description

Compose the result of the instruction from the odd-numbered half-words obtained from the two source operands. This instruction does not modify any flags.

Pseudo Code

```
a = (c.h3 << 48) | (c.h1 << 32) | (b.h3 << 16) | b.h1
```

Assembly Code Example

```
VPACK4HM r0,r1,r2     ;; pack odd half-words of r1 and r2 into 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 Table 8-7 and Table 8-8.

		Instruction Code
VPACK4HM	a,b,c	01011bbb00110001BBBCCCCCCAAAAAA
VPACK4HM	a,b,u6	01011bbb01110001BBBuuuuuuAAAAAA
VPACK4HM	b,b,s12	01011bbb10110001BBBssssssSSSSSS
VPACK4HM<.cc>	b,b,c	01011bbb11110001BBBCCCCCC0QQQQQ
VPACK4HM<.cc>	b,b,u6	01011bbb11110001BBBuuuuuu1QQQQQ

VPACK2WL

Function

Compose the destination result using the even-numbered words from the source operands

Extension Group:

Baseline

Operation:

```
a = (c.w0 << 32) | b.w0
```

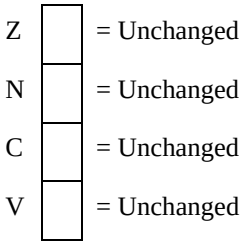
Instruction Format

```
op a, b, c
```

Syntax Example

```
VPACK2WL a, b, c
```

STATUS32 Flags Affected:



Description

Compose the result of the instruction from the even-numbered word (w0) obtained from each of the two source operands. This instruction does not modify any flags.

Pseudo Code

```
a = (c.w0 << 32) | b.w0
```

Assembly Code Example

```
VPACK2WL r0,r1,r2     ;; pack even (i.e. lower) words of r1 and r2 into 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 Table 8-7 and Table 8-8.

		Instruction Code
VPACK2WL	a,b,c	01011bbb00110010 1BBBCCCCCCAAAAAA
VPACK2WL	a,b,u6	01011bbb01110010 1BBBuuuuuuAAAAAA
VPACK2WL	b,b,s12	01011bbb10110010 1BBBssssssSSSSSS
VPACK2WL<.cc>	b,b,c	01011bbb11110010 1BBBCCCCCC0QQQQQ
VPACK2WL<.cc>	b,b,u6	01011bbb11110010 1BBBuuuuuu1QQQQQ

VPACK2WH

Function

Compose the destination result using the odd-numbered words from the source operands

Extension Group:

Baseline

Operation:

```
a = (c.w1 << 32) | b.w1
```

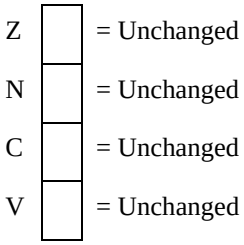
Instruction Format

```
op a, b, c
```

Syntax Example

```
VPACK2WH a, b, c
```

STATUS32 Flags Affected:



Description

Compose the result of the instruction from the odd-numbered word obtained from each of the two source operands. This instruction does not modify any flags.

Pseudo Code

```
a = (c.w1 << 32) | b.w1
```

Assembly Code Example

```
VPACK2WL r0,r1,r2     ;; pack odd (i.e. higher) words of r1 and r2 into 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 Table 8-7 and Table 8-8.

		Instruction Code
VPACK2WH	a,b,c	01011bbb00110011BBBCCCCCCAAAAAA
VPACK2WH	a,b,u6	01011bbb01110011BBBuuuuuuAAAAAA
VPACK2WH	b,b,s12	01011bbb10110011BBBssssssSSSSSS
VPACK2WH<.cc>	b,b,c	01011bbb11110011BBBCCCCCC0QQQQQ
VPACK2WH<.cc>	b,b,u6	01011bbb11110011BBBuuuuuu1QQQQQ

VPACK2WM

Function

Compose the destination result using the odd-numbered words from the source operands

Extension Group:

Baseline

Operation:

```
a = (c.w1 << 32) | b.w1
```

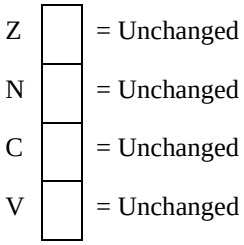
Instruction Format

```
op a, b, c
```

Syntax Example

```
VPACK2WM a, b, c
```

STATUS32 Flags Affected:



Description

Compose the result of the instruction from the odd-numbered word (w1) obtained from each of the two source operands. This instruction does not modify any flags.

Pseudo Code

```
a = (c.w1 << 32) | b.w1
```

Assembly Code Example

```
VPACK2WM r0,r1,r2     ;; pack odd (i.e. upper) words of r1 and r2 into 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 Table 8-7 and Table 8-8.

		Instruction Code
VPACK2WM	a,b,c	01011bbb00110100 1BBBCCCCCCAAAAAA
VPACK2WM	a,b,u6	01011bbb01110100 1BBBuuuuuuAAAAAA
VPACK2WM	b,b,s12	01011bbb10110100 1BBBssssssSSSSSS
VPACK2WM<.cc>	b,b,c	01011bbb11110100 1BBBCCCCCC0QQQQQ
VPACK2WM<.cc>	b,b,u6	01011bbb11110100 1BBBuuuuuu1QQQQQ

VMAC2H

This section describes the standard MPYD instruction.

Function

Signed multiplication and accumulation of two 16-bit vectors.

Extension Group

Baseline

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 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 Table 8-7 and Table 8-8.

		Instruction Code
VMAC2H	a,b,c	00101bbb00011110 0BBBCCCCCCAAAAAA
VMAC2H	a,b,u6	00101bbb01011110 0BBBuuuuuuAAAAAA
VMAC2H	b,b,s12	00101bbb10011110 0BBBssssssSSSSSS
VMAC2H<.cc>	b,b,c	00101bbb11011110 0BBBCCCCCC0QQQQQ
VMAC2H<.cc>	b,b,u6	00101bbb11011110 0BBBuuuuuu1QQQQQ

VMAX2

Function

Compare two-way 32-bit vectors and return the maximum

Extension Group

ARC64: baseline

Operation

```
A.w1 = max (B.w1, C.w1)
A.w0 = max (B.w0, C.w0)
```

Instruction Format

op a, b, c

Syntax Example

VMAX2<.cc> a, b, c

VMAX2 a, b, LIMM

STATUS32 Flags Affected

None

Description

Compare the 32-bit signed vector elements of the 64-bit operands b and c. The maximum value of the two elements is stored in the corresponding element of the destination operand. This instruction does not update any STATUS32 flags. **Pseudo Code**

```
if (cc == true) {                                        /* VMAX2*/
  bl = b & 0x0000_0000_ffff_ffff;
  cl = c & 0x0000_0000_ffff_ffff;
  bh = (b >> 32) & 0x0000_0000_ffff_ffff;
  ch = (c >> 32) & 0x0000_0000_ffff_ffff;
  al = bl >= cl ? bl : cl;
  ah = bh >= ch ? bh : ch;
  a = (ah << 32) | al;
}
```

Assembly Code Examples

```
VMAX2    r0, r1, r2     ; Compare the 32-bit vector elements of the
                        ; operands r1 and r2, and return their maximum
                        ; elements in r0.
```

Instruction Encodings

		Instruction Code
VMAX2	a,b,c	00101bbb00111001 1BBBCCCCCCAAAAAA
VMAX2	a,b,u6	00101bbb01111001 1BBBuuuuuuAAAAAA
VMAX2	b,b,s12	00101bbb10111001 1BBBssssssSSSSSS
VMAX2<.cc>	b.b,c	00101bbb11111001 1BBBCCCCCC0QQQQQ
VMAX2<.cc>	b,b,u6	00101bbb11111001 1BBBuuuuuu1QQQQQ

VMIN2

Function

Compare two-way 32-bit vectors and return the minimum

Extension Group

ARC64: baseline

Operation

```
A.w1 = min (B.w1, C.w1)
A.w0 = min (B.w0, C.w0)
```

Instruction Format

op a, b, c

Syntax example

VMIN2<.cc> a, b, c

VMIN2 a, b, LIMM

STATUS32 Flags Affected

None

Description:

Compare the 32-bit signed vector elements of the 64-bit operands b and c. The minimum value of the two elements is stored in the corresponding element of the destination operand. This instruction does not update any STATUS32 flags. **Pseudo Code**

```
if (cc == true) {                                        /* VMIN2*/
  bl = b & 0x0000_0000_ffff_ffff;
  cl = c & 0x0000_0000_ffff_ffff;
  bh = (b >> 32) & 0x0000_0000_ffff_ffff;
  ch = (c >> 32) & 0x0000_0000_ffff_ffff;
  al = bl < cl ? bl : cl;
  ah = bh < ch ? bh : ch;
  a = (ah << 32) | al;
}
```

Assembly Code Examples

```
VMIN2    r0, r1, r2     ; Compare the 32-bit vector elements of the
                        ; operands r1 and r2, and return their minimum
                        ; elements in r0.
```

Instruction Encodings

		Instruction Code
VMIN2	a,b,c	00101bbb00111000 1BBBCCCCCCAAAAAA
VMIN2	a,b,u6	00101bbb01111000 1BBBuuuuuuAAAAAA
VMIN2	b,b,s12	00101bbb10111000 1BBBssssssSSSSSS
VMIN2<.cc>	b.b,c	00101bbb11111000 1BBBCCCCCC0QQQQQ
VMIN2<.cc>	b,b,u6	00101bbb11111000 1BBBuuuuuu1QQQQQ

VMAC2HU

This section describes the standard MPYD instruction.

Function

Unsigned multiplication and accumulation of two 16-bit vectors.

Extension Group

Baseline

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 and should be an even-numbered register. This instruction does not modify any flags.

Pseudo Code

```
if (cc) {                                                  /* VMAC2HU*/
  A.w0 = (acc.w0 += (b.h0 * c.h0));
  A.w1 = (acc.w1 += (b.h1 * c.h1));
}
```

Assembly Code Example

```
VMAC2HU r0,r2,r3      ;Unsigned integer multiply two vectors r2 and r3 and
                      ; return result in r0 and r1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VMAC2HU	a,b,c	00101bbb00011111 0BBBCCCCCCAAAAAA
VMAC2HU	a,b,u6	00101bbb01011111 0BBBuuuuuuAAAAAA
VMAC2HU	b,b,s12	00101bbb10011111 0BBBssssssSSSSSS
VMAC2HU<.cc>	b,b,c	00101bbb11011111 0BBBCCCCCC0QQQQQ
VMAC2HU<.cc>	b,b,u6	00101bbb11011111 0BBBuuuuuu1QQQQQ

VMPY2H

This section describes the standard MPYD instruction.

Function

Dual 16-bit SIMD multiplication

Extension Group

Baseline

Operation

```
if (cc) {
   acc.w0 = A.w0 = b.h0 * c.h0;
   acc.w1 = A.w1 = b.h1 * c.h1;
}
```

Note `wo` (lower) and `w1` (upper) represent the 32-bit elements of a 64-bit operand.

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 Table 8-7 and Table 8-8.

		Instruction Code
VMPY2H	a,b,c	00101bbb00011100 0BBBCCCCCCAAAAAA
VMPY2H	a,b,u6	00101bbb01011100 0BBBuuuuuuAAAAAA
VMPY2H	b,b,s12	00101bbb10011100 0BBBssssssSSSSSS
VMPY2H<.cc>	b,b,c	00101bbb11011100 0BBBCCCCCC0QQQQQ
VMPY2H<.cc>	b,b,u6	00101bbb11011100 0BBBuuuuuu1QQQQQ

VMPY2HU

This section describes the standard MPYD instruction.

Function

Dual unsigned 16-bit SIMD multiplication

Extension Group

Baseline

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. `wo` (lower) and `w1` (upper) represent the 32-bit elements of a 64-bit operand.

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 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 Table 8-7 and Table 8-8.

		Instruction Code
VMPY2HU	a,b,c	00101bbb00011101 0BBBCCCCCCAAAAAA
VMPY2HU	a,b,u6	00101bbb01011101 0BBBuuuuuuAAAAAA
VMPY2HU	b,b,s12	00101bbb10011101 0BBBssssssSSSSSS
VMPY2HU<.cc>	b,b,c	00101bbb11011101 0BBBCCCCCC0QQQQQ
VMPY2HU<.cc>	b,b,u6	00101bbb11011101 0BBBuuuuuu1QQQQQ

VSUB2

Function

Dual 32-bit SIMD subtract.

Extension Group

Baseline

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.

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 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 Table 8-7 and Table 8-8.

		Instruction Code
VSUB2	a,b,c	00101bbb00111010BBBCCCCCCAAAAAA
VSUB2	a,b,u6	00101bbb01111010BBBuuuuuuAAAAAA
VSUB2	b,b,s12	00101bbb10111010BBBssssssSSSSSS
VSUB2<.cc>	b,b,c	00101bbb11111010BBBCCCCCC0QQQQQ
VSUB2<.cc>	b,b,u6	00101bbb11111010BBBuuuuuu1QQQQQ

VSUB2H

Function

Dual 16-bit vector subtraction.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
VSUB2H	a,b,c	00101bbb00010101 0BBBCCCCCCAAAAAA
VSUB2H	a,b,u6	00101bbb01010101 0BBBuuuuuuAAAAAA
VSUB2H	b,b,s12	00101bbb10010101 0BBBssssssSSSSSS
VSUB2H<.cc>	b,b,c	00101bbb11010101 0BBBCCCCCC0QQQQQ
VSUB2H<.cc>	b,b,u6	00101bbb11010101 0BBBuuuuuu1QQQQQ

VSUB4H

Function

Quad 16-bit SIMD subtraction.

Extension Group

Baseline

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.

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 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 Table 8-7 and Table 8-8.

		Instruction Code
VSUB4H	a,b,c	00101bbb00111001 0BBBCCCCCCAAAAAA
VSUB4H	a,b,u6	00101bbb01111001 0BBBuuuuuuAAAAAA
VSUB4H	b,b,s12	00101bbb10111001 0BBBssssssSSSSSS
VSUB4H<.cc>	b,b,c	00101bbb11111001 0BBBCCCCCC0QQQQQ
VSUB4H<.cc>	b,b,u6	00101bbb11111001 0BBBuuuuuu1QQQQQ

VSUBADD

Function

Dual 32-bit SIMD subtract and add.

Extension Group

Baseline

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.

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, 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 Table 8-7 and Table 8-8.

		Instruction Code
VSUBADD	a,b,c	00101bbb00111111 0BBBCCCCCCAAAAAA
VSUBADD	a,b,u6	00101bbb01111111 0BBBuuuuuuAAAAAA
VSUBADD	b,b,s12	00101bbb10111111 0BBBssssssSSSSSS
VSUBADD<.cc>	b,b,c	00101bbb11111111 0BBBCCCCCC0QQQQQ
VSUBADD<.cc>	b,b,u6	00101bbb11111111 0BBBuuuuuu1QQQQQ

VSUBADD2H

Function

Dual 16-bit vector subtraction and addition.

Extension Group

Baseline

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. .

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 Table 8-7 and Table 8-8.

		Instruction Code
VSUBADD2H	a,b,c	00101bbb00010111 0BBBCCCCCCAAAAAA
VSUBADD2H	a,b,u6	00101bbb01010111 0BBBuuuuuuAAAAAA
VSUBADD2H	b,b,s12	00101bbb10010111 0BBBssssssSSSSSS
VSUBADD2H<.cc>	b,b,c	00101bbb11010111 0BBBCCCCCC0QQQQQ
VSUBADD2H<.cc>	b,b,u6	00101bbb11010111 0BBBuuuuuu1QQQQQ

VSUBADD4H

Function

Quad16-bit SIMD subtract and add.

Extension Group

Baseline

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.

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 Table 8-7 and Table 8-8.

		Instruction Code
VSUBADD4H	a,b,c	00101bbb00111011 0BBBCCCCCCAAAAAA
VSUBADD4H	a,b,u6	00101bbb01111011 0BBBuuuuuuAAAAAA
VSUBADD4H	b,b,s12	00101bbb10111011 0BBBssssssSSSSSS
VSUBADD4H<.cc>	b,b,c	00101bbb11111011 0BBBCCCCCC0QQQQQ
VSUBADD4H<.cc>	b,b,u6	00101bbb11111011 0BBBuuuuuu1QQQQQ

WAIT

Function

Wait for a wake-up event of any type. Enter sleep state and wait for a wakeup event

Extension Group

Baseline

Operation

DEBUG[ZZ] = 1;

Instruction Format

op c

Syntax Example

WAIT

STATUS32 Flags Affected

IE	•	Updated depending on the SLEEP operand as explained in Table 11-10
E[3:0]	•	Updated depending on the SLEEP operand as explained in Table 11-10
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
ZZ	•	= 1
SM	•	Updated depending on the SLEEP operand as explained in Table 11-10

Description

If the lock flag is set, this instruction causes the processor to enter the sleep state and remain asleep until woken up by one of the following events:

- The lock flag 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 external event input signal transitions from 0 to 1.
- An interrupt is taken.
- An external halt request is acknowledged by the processor.

- A debug transaction places the processor in a halt state.

If the lock flag is clear, this instruction behaves as a NOP. Hence, the WAIT instruction is similar to the WLFC instruction, however, it includes the external event input as an additional wake-up condition.

Similar to WLFC, the WAIT instruction is available in both kernel and User modes provided User mode sleep is enabled by the STATUS32.US bit. If User mode sleep is disabled (that is, STATUS32.US == 0), then the processor raises an `EV_PrivilegeV` exception if this instruction is attempted regardless of the state of the lock flag.

The WAIT instruction has identical operands to the SLEEP instruction. For more information on how the SLEEP instruction operands work, see "SLEEP instruction operands" .

Pseudo Code

```
WAIT (operand)
      if ((STATUS32.U == 1) || STATUS32.US == 0))
      raiseException (EV_PrivilegeV)
else
if (LF ==1)
         Sleep operand
```

Assembly Code Example

The WAIT instruction can be placed anywhere in the instruction sequence.

```
WAIT 0 ;; enter sleep without changing interrupt enable
```

Instruction Encodings

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
WAIT	c	00100000001011110000cccccc111111
WAIT	<u6>	00100000011011110000uuuuuu111111

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 11-10
E[3:0]	•	Updated depending on the SLEEP operand as explained in Table 11-10
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
ZZ	•	= 1
SM	•	Updated depending on the SLEEP operand as explained in Table 11-10

Description

The ARCv3-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 11-7 WEVT placement in code

```
SUB r2, r2, 0x1
ADD r1, r1, 0x2
WEVT
...
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
WEVT	c	00100000001011111000CCCCCC111111
WEVT	<u6>	00100000011011111000uuuuuu111111

WLFC

Function

Enter the sleep state to reduce dynamic power during busy-waiting loops

Extension Group

Baseline

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 11-10
E[3:0]	•	Updated depending on the SLEEP operand as explained in Table 11-10
Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
ZZ	•	= 1
SM	•	Updated depending on the SLEEP operand as explained in Table 11-10

Description

The ARCv3-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 11-8 WLFC Placement in Code

```
SUB r2, r2, 0x1
ADD r1, r1, 0x2
WLFC
...
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
WLFC	c	00100001001011111000 1CCCCCC111111
WLFC	<u6>	00100001011011111000 1uuuuuu111111

XBFU

Function

Extract unsigned bitfield

Extension Group

Baseline

Operation

if (cc) then a = (b >> c[4:0]) & ((1 << (c[9:5]+1)) - 1)

Instruction Format

op b, b, u5, u5

op a, b, c

op a, b, LIMM

op a, LIMM, b

Syntax Example

XBFU<.f> b, b, u5, u5

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Source operand 1 specifies a 32-bit value, from which a bit-field is extracted and assigned to the destination operand. The extracted bit-field starts at bit *N* and includes *M* bits of operand 1.

The value of N is specified by the unsigned binary value obtained from the least-significant five bits of operand 2. M is obtained by 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

```
                                        /* XBFU
N = src2[4:0]                           *first bit to extract
M = src2[9:5] + 1                       *number of bits to
if cc==true then                        *extract
 dest = (src1 >> N) && ((1 << M)-1)     */
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
```

Assembly Code Example

```
XBFU r1,r1,4,6       ;extract bits 4 through 9 from r1 into r1

XBFU r2,r2,0,5       ; extract bits 0 thru 5 from r2 into r2
XBFU r1,r2,r3        ;extract bits from r2 depending on r3
XBFU.F 0,r2,3,5      ;set Z flag if bits 3 to 7 in r2 are zero
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
XBFU<.f>	a,b,c	00100bbb00101101FBBBCCCCCCAAAAAA
XBFU<.f>	a,b,u6	00100bbb01101101FBBBuuuuuuAAAAAA
XBFU<.f>	b,b,s12	00100bbb10101101FBBBssssssSSSSSS
XBFU<.cc><.f>	b,b,c	00100bbb11101101FBBBCCCCCC0QQQQQ
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 LIMM value as the second source operand, that is, using format `a,b,c` where `c` is a LIMM.

An XBFU instruction that extracts a field that is less than or equal to two bits can be encoded using a u6 operand. In this case, only bit 0 of the M operand can be specified, and bits 1 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.

XBFUL

Function

Extract unsigned bitfield from long

Extension Group

Baseline

Operation

if (cc) then a = (b >> c[5:0]) & ((1 << (c[11:6]+1)) - 1)

Instruction Format

op b, b, u6, u6

op a, b, c

op a, b, LIMM

op a, LIMM, b

Syntax Example

XBFU<.f> b, b, u5, u5

STATUS32 Flags Affected

Z	•	= Set if result is zero
N		= Unchanged
C		= Unchanged
V		= Unchanged

Description

Source operand 1 specifies a 64-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 six bits of operand 2. M is obtained by adding 1 to the unsigned six-bit binary value obtained from bits [11:6] 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

```
                                        /* XBFU
N = src2[5:0]                           *first bit to extract
M = src2[11:6] + 1                      *number of bits to
if cc==true then                        *extract
 dest = (src1 >> N) && ((1 << M)-1)     */
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
```

Assembly Code Example

```
XBFUL r1,r1,4,6      ;extract bits 4 through 9 from r1 into r1
XBFUL r2,r2,0,5
XBFUL r1,r2,r3       ;extract bits 0 thru 5 from r2 into r2
XBFUL.F 0,r2,3,5     ;extract bits from r2 depending on r3
                     ;set Z flag if bits 3 to 7 in r2 are zero
```

Syntax and Encoding

		Instruction Code
XBFUL<.f>	a,b,c	01011bbb00101101FBBBCCCCCCAAAAAA
XBFUL<.f>	a,b,u6	01011bbb01101101FBBBuuuuuuAAAAAA
XBFUL<.f>	b,b,s12	01011bbb10101101FBBBssssssSSSSSS
XBFUL<.cc><.f>	b,b,c	01011bbb11101101FBBBCCCCCC0QQQQQ
XBFUL<.cc><.f>	b,b,u6	01011bbb11101101FBBBuuuuuu1QQQQQ

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 6-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 6-bit offsets of the XBFUL instruction can be encoded using a LIMM value as the second source operand, that is, using format `a,b,c` where `c` is a LIMM.

An XBFUL instruction that extracts a 1-bit field can be encoded using a u6 operand. In this case, the M operand is 1, which is encoded as zero, and thus the extracted field length is 1. The N operand can be fully-specified with a u6 operand. This format allows the source and destination operand registers to be distinct.

XOR

Function

Bitwise Exclusive OR

Extension Group

Baseline

Operation

if (cc) a = b ^ 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 Table 8-7 and Table 8-8.

		Instruction Code
XOR<.f>	a,b,c	00100bbb00000111FBBBCCCCCCAAAAAA
XOR<.f>	a,b,u6	00100bbb01000111FBBBuuuuuuAAAAAA
XOR<.f>	b,b,s12	00100bbb10000111FBBBssssssSSSSSS
XOR<.cc><.f>	b,b,c	00100bbb11000111FBBBCCCCCC0QQQQQ
XOR<.cc><.f>	b,b,u6	00100bbb11000111FBBBuuuuuu1QQQQQ
XOR<.aq.rl>	b,[c]	00100bbb00101111mBBBCCCCCC110011
XOR_S	b,b,c	01111bbbccc00111

XORL

Function

Bitwise Exclusive OR Long

Extension Group

Baseline

Operation

if (cc) a = b ^ c;

Instruction Format

op a, b, c

Syntax Example

XORL<.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 64-bit source operand 1 (b) with 64-bit source operand 2 (c). The 64-bit 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                        /* XORL */
 dest = src1 XOR src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[63]
```

Assembly Code Example

```
XORL r1,r2,r3   ; Logical XOR r2 with r3 and write 64-bit result into r1
```

Syntax and Encoding

		Instruction Code
XORL<.f>	a,b,c	01011bbb00000111FBBBCCCCCCAAAAAA
XORL<.f>	a,b,u6	01011bbb01000111FBBBuuuuuuAAAAAA
XORL<.f>	b,b,s12	01011bbb10000111FBBBssssssSSSSSS
XORL<.cc><.f>	b,b,c	01011bbb11000111FBBBCCCCCC0QQQQQ
XORL<.cc><.f>	b,b,u6	01011bbb11000111FBBBuuuuuu1QQQQQ

12 The Host

12.1 The Host Interface

The ARCv3-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 ARCv3-based processor using special registers. How the various parts of the ARCv3-based processor appear to the host depends on the host interface.

This section provides an overview of the techniques to control ARCv3-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 ARCv3-based processor appear as a memory mapped section to the host. Figure 12-1 and Figure 12-2 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 12-1 Example Host Memory Maps, Contiguous Host Memory

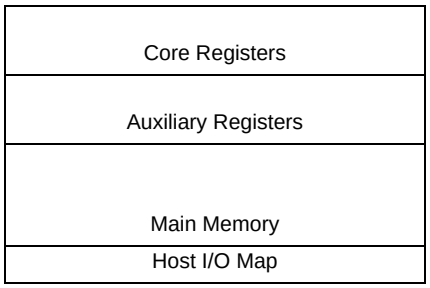
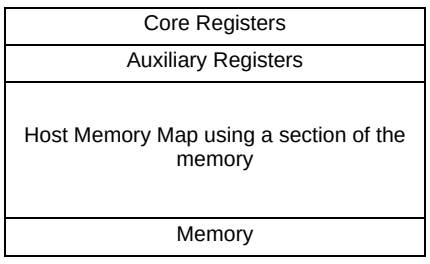


Figure 12-2 Example Host Memory Maps, Host Memory and Host I/O



Once a Reset has occurred, the ARCv3-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 ARCv3-based processor, depending on whether the ARCv3-based processor is running or halted as shown in Table 12-1 on page 864.

Table 12-1 Host Accesses to the ARCv3-based processor

	Running	Halted
Memory	Read/Write	Read/Write
Auxiliary Registers	Read/Write	Read/Write
Core Registers	Read/Write	Read/Write

12.2 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).

12.3 Memory

The program memory may be changed by the host. The memory can be changed at any time by the host.

If program code is being altered, or transferred into ARCv3-based memory space, the instruction cache must be invalidated.

12.4 Halting

The ARCv3-based processor can halt itself with the FLAG instruction or the host can halt the processor. The host halts the ARCv3-based processor by setting the FH bit in the Debug Register, DEBUG register.

To force the ARCv3-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 ARCv3-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 ARCv3-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 ARCv3-based processor.

12.5 Starting

The host starts the ARCv3-based processor by clearing the H bit in the Status Register, STATUS32 register.

If the ARCv3-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 ARCv3-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 ARCv3-based processor continues from where it left off when it was stopped.

12.6 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 12-2 on page 865).

Table 12-2 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.

12.6.1 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 ARCv3-based processor wakes up at the next single-step.

See "SLEEP" on page 733 and "Null Instruction Format" on page 108 for further details.

12.6.2 BRK Instruction in Single Instruction Step Mode

The BRK instruction behaves exactly as when the processor is not in the Single Step Mode.

12.7 Software Breakpoints

The BRK instruction can also be used to insert a software breakpoint. BRK halts the ARCv3-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:

- "Memory Protection Unit"
- "Protection Schemes"
- "Memory Management Unit"
- "ICCM"
- "Instruction Cache"
- "DCCM"
- "Data Cache"
- "Error Protection"
- "Peripheral Bus"
- "Write Buffer"
- "Branch Prediction Unit"
- "External Host Debugging"
- "Debug Features"
- "Performance Counters"
- "Processor Timers""ARC Trace"
- "Cluster Network"

- "Hardware Prefetching"
- "Power Management Features"
- "ARConnect"
- "Cluster DMA Controller"
- "ROM Patching Unit"

13 Memory Protection Unit

13.1 Memory Protection Unit

The ARCv3-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 ARCv3-based processors raises a Protection Violation exception, and this exception prevents the faulting instruction from completing.

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. The permission checks in the MPU region are independent for user and kernel modes. A privilege exception is raised when:

- In Kernel mode you try to access resources with user permissions
- In User mode you try to access resources with kernel permissions

You can define default kernel and user permissions for accesses that fall outside all memory protection regions.

Multiple MPU regions are allowed to overlap and the attributes that apply to a particular address in the case of overlapping regions are determined using a fixed priority scheme.

13.1.1 Key Features

The following are the main features of ARCv3-based processor MPU:

- Programmable execute permission bits to enable or disable execution of code from specific regions of memory.
- Programmable data read and write permission bits to enable or disable data access to specific regions of memory.
- Separate kernel and user mode read, write, and execute permissions.
- 1, 2, 4, 8....32 configurable memory regions.
- Regions can be programmed individually and independently.
- Overlapping regions are supported by a priority scheme.
- Ability to set default permissions that apply to accesses outside all programmed protection regions.
- Uses EV_ProtV (see Exception Vectors and the Exception Cause Register on page 199) exception to indicate access violations.
- Protection exceptions are precise and can be restarted.
- Can be used in conjunction with the Stack Checking mechanisms in the ARCv3-based processors.
- Programmable cacheability attribute per MPU region
- Programmable memory attributes per MPU region

13.1.2 Configuration Options

The ARCv3MPU is an optional component for the ARCv3-based processors. The configuration option, MPU_NUM_REGIONS, defines the number of regions supported by the MPU.

13.1.3 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.

13.1.4 Data Organization and Addressing

The ARCv3 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 MPU region. The maximum number of protection regions is fixed at configuration time and is specified in the MPU_BUILD register (Memory Protection Build Configuration Register, MPU_BUILD).

Protection regions are allowed to overlap each other: the attributes that apply to a particular address in the case of overlapping regions are determined using a fixed priority scheme. This mechanism can be used to implement a flexible organization based on need, for example, a large read-only region with a smaller embedded writeable region.

Each region has a base address and size, specified in the MPU Region Descriptor Base Registers, MPU_RDB0 to MPU_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`.

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:

13.1.4.1 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.

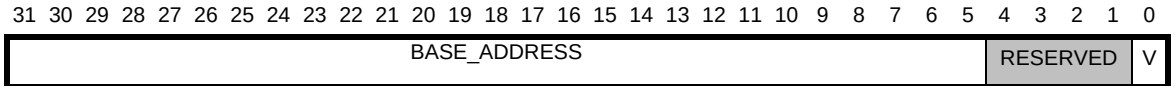
Example 1 Memory Map shows the memory map for this set up:

Table 13-1 Example 1 -- Contiguous Regions Memory Map

Region 0	0x00000000	Base address: 0x00000000
		Region size: 2G
	0x7FFFFFFF	Regions permissions: user-execute
Region 1	0x80000000	Base address: 0x80000000
		Region size: 1G
	0xBFFFFFFF	Regions permissions: user-read, user-write
Region 2	0xC0000000	Base address: 0xC0000000
		Region size: 1G
	0xFFFFFFFF	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 13-1 Example 1: MPU_RDB0 Register



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 13-2 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 13-3 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 13-4 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 13-5 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 13-6 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

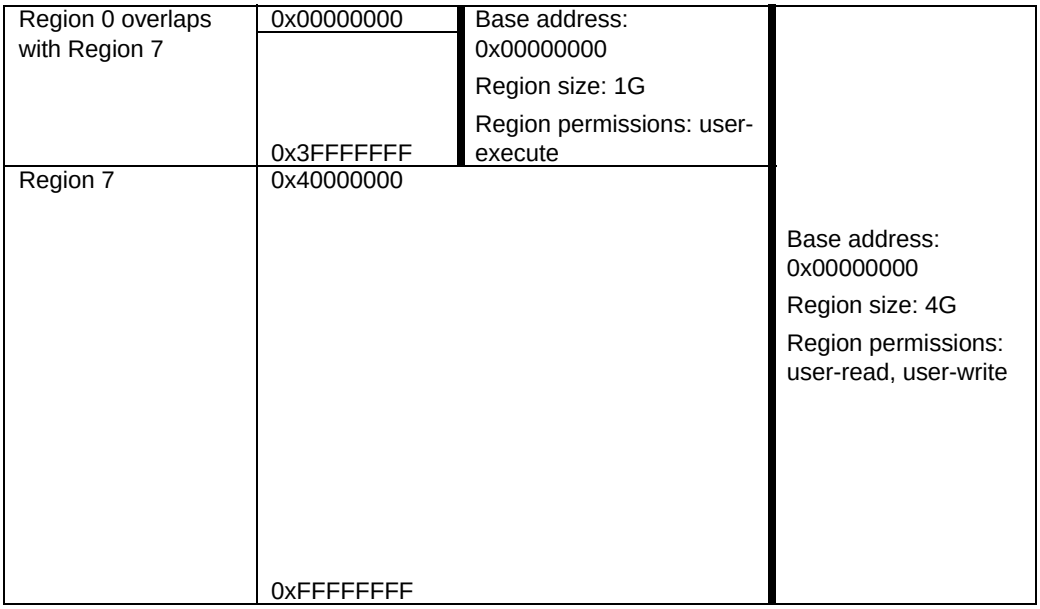
13.1.4.2 Example 2 - Overlapping Regions

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 13-7 Overlapping Regions Memory Map



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 13-8 Example 2: MPU_RDB0 Register

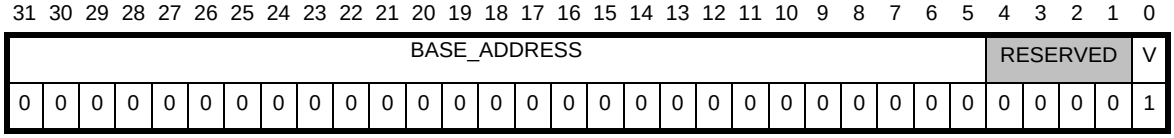


Figure 13-9 Example 2: MPU_RDP0 Register

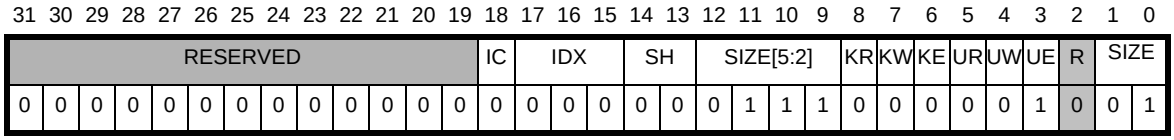
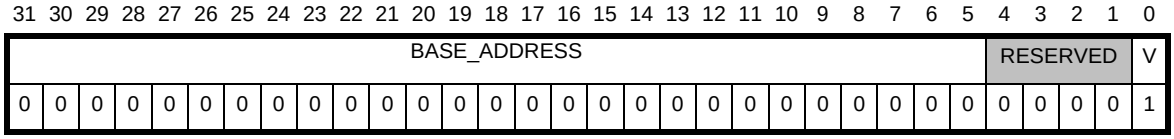


Figure 13-10 Example 2: MPU_RDB7 Register



13.1.5 Modes of Operation

After reset, the ARCv3 registers contain default values that disable the ARCv3 MPU. All the region descriptor registers contain zero.

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

13.1.5.1 Region Accesses

All accesses are checked against the region descriptors, according to a fixed priority, to determine what attributes to apply.

Table 13-2 Priority Configurations

Region	Priority (8-region configuration)	Priority (16-region configuration)	Priority (32-region configuration)
0	Highest	Highest	Highest
1	2nd Highest	2nd Highest	2nd Highest
2	3rd Highest	3rd Highest	3rd Highest
3	4th Highest	4th Highest	4th Highest
4	5th Highest	5th Highest	5th Highest
5	6th Highest	6th Highest	6th Highest
6	7th Highest	7th Highest	7th Highest
7	Lowest	8th Highest	8th Highest
8	Not applicable	9th Highest	9th Highest
9	Not applicable	10th Highest	10th Highest
10	Not applicable	11th Highest	11th Highest
11	Not applicable	12th Highest	12th Highest
12	Not applicable	13th Highest	13th Highest
13	Not applicable	14th Highest	14th Highest
14	Not applicable	15th Highest	15th Highest
15	Not applicable	Lowest	16th Highest
16	Not applicable	Not applicable	17th Highest
17	Not applicable	Not applicable	18th Highest
18	Not applicable	Not applicable	19th Highest
19	Not applicable	Not applicable	20th Highest
20	Not applicable	Not applicable	21th Highest

Region	Priority (8-region configuration)	Priority (16-region configuration)	Priority (32-region configuration)
21	Not applicable	Not applicable	22th Highest
22	Not applicable	Not applicable	23th Highest
23	Not applicable	Not applicable	24th Highest
24	Not applicable	Not applicable	25th Highest
25	Not applicable	Not applicable	26th Highest
26	Not applicable	Not applicable	27th Highest
27	Not applicable	Not applicable	28th Highest
28	Not applicable	Not applicable	29th Highest
29	Not applicable	Not applicable	30th Highest
30	Not applicable	Not applicable	31th 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.

13.1.5.2 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 13-3 lists the settings of the ECR register based on the entry to the EV_ProtV exception.

Table 13-3 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	0x30	0x06	0x01	0x04	0x060104

Failed Reference type	Vector Name	Vector Offset	Vector Number	Cause Code	Parameter	ECR value
Memory Write (such as ST, PUSH, ENTER, interrupt entry, SCOND)	EV_ProtV	0x30	0x06	0x02	0x04	0x060204
Memory Read-Modify-Write (such as EX)	EV_ProtV	0x30	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 it's delay slot always reside within a single MPU region, or in regions that have the same set of access permissions.

13.1.5.3 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 ARCv3-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 ARCv3 architecture gives higher priority to instruction fetch violations.

Due to the pipelined construction of the ARCv3-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 ARCv3 architecture: 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 ARCv3-based processor with MPU and Stack Checking, and one Load or Store instruction to violate the protection policies of all two 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 13-4 ECR parameter values for EV_ProtV sources

Source of EV_ProtV	ECR Parameter Value
Stack Checking	0x02
MPU	0x04
MMU	0x08

Therefore, if the address of a Load instruction is detected by the MPU as a protection violation, as well as having an address that is outside the defined stack region, the ECR parameter value is 0x6 (MPU and Stack Checking bits all set).

However, if the same instruction also raises a protection violation due to its PC value being within a region that does not have execute permission, the instruction violation takes precedence over all data violations, from whatever source, and thus the ECR parameter value is 0x4 (MPU only).

13.1.5.4 Caution When Updating Region Settings

As the MPU protects all the memory space on ARCv3-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 into the region descriptors when they are enabled. A disabled region descriptor never triggers a protection violation, and can be programmed with illegal values without causing problems.

13.1.6 Memory Protection Unit Registers

ARCv3-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 13-5 lists the MPU auxiliary registers.

Table 13-5 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 13-5 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 13-5 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

13.1.7 MPU Enable Register, MPU_EN

Address: 0x409

Access: RW

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 13-11 MPU_EN Register

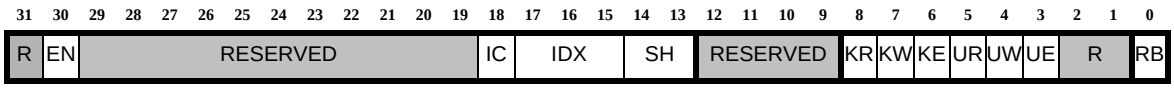


Table 13-6 MPU Enable Register

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 >= 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 ■ `1`: Instruction fetch is permitted when in user
UW	[4]	General Write Permission ■ `0`: Region is write-protected ■ `1`: Region is writeable when in user

Table 13-6 MPU Enable Register

Field	Bit	Description
UR	[5]	General Read Permission ■ 0: Region is read-protected. ■ 1: Region is readable when in user 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 ■ 1: Instruction fetch is permitted granted when in kernel mode only Note that user mode execution does not imply kernel-mode write.
KW	[7]	Kernel-only Write Permission ■ 0: Region is write-protected ■ 1: Region is writeable when in kernel mode only Note that user mode write does not imply kernel-mode write.
KR	[8]	Kernel-only Read Permission ■ 0: Region is read-protected. ■ 0: 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. User mode read does not imply kernel-mode read.
SH	[14:13]	Shareability attribute applied to this region: ■ 0x0: Non-shareable. Data in this region can be cached, but is considered private, and hence no snooping occurs to addresses in this region ■ 0x1: Reserved. Treated as inner-shareable ■ 0x2: Outer-shareable. Data in this region is considered to be shareable across the system. The region is included in the global coherency domain, and snooping requests are propagated through the inter-cluster interconnect to maintain global coherency ■ 0x3 Inner-shareable. Data in this region is considered to be shareable within the local cluster. The region is included in the local (or inner) coherency domain, and snooping is limited to this inner domain
IDX	[17:15]	This 3-bit field is a pointer to a field in the memory attribute auxiliary register where the memory attributes are specified. These attributes are applicable to data accesses (LD/ST) and not for Instruction fetch. See MPU_MEM_ATTR register.

Table 13-6 MPU Enable Register

Field	Bit	Description
IC	[18]	Code cacheability for the default MPU region ■ `0x0`: Code is cacheable in all levels of the cluster cache hierarchy ■ `0x1`: Code is not cacheable in any level of the cluster cache hierarchy (L1, L2, and so on) Notes: ■ This bit is valid only when the MPU version >= 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 uncacheable 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_EN[12]==1).
EN	[30]	MPU enable bit A value of 0 means disabled; 1 means enabled.

13.1.8 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. These upper 32 bits shall be IOW / RAZ when accessed using 64-bit auxiliary memory access instructions (LRL, SRL, AEXL).

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 and Stack out-of-bounds, etc). The MR field indicates the actual region selected by a faulting address. The special value 0xff for this MR field indicates that no region match was detected.

Figure 13-12 MPU_ECR Register

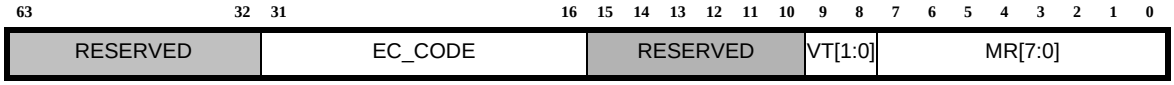


Table 13-7 MPU Exception Cause Register

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 13-7 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: ■ `00`: Execution Violation ■ `01`: Data Read Violation ■ `10`: Data Write Violation ■ `11`: 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

13.1.9 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_RDB1: 0x424 ....
MPU_RDB31: 0x440

Access: RW

Reset 0x00000000

The MPU region descriptor base registers contain the base address for each region and enable 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.

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 13-13 MPU Region Descriptor Base Registers

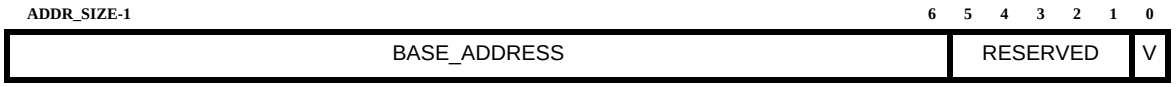


Table 13-8 MPU Region Descriptor Base Registers Field Description

Field	Bit	Description
V	[0]	Region Descriptor Validity Flag ■ `0`: This region descriptor is unused. ■ `1`: This region descriptor is valid and permissions defined by the corresponding MPU_RDP register will be applied to each memory reference according to the type of the reference.
BASE_ADDRESS	[ADDR_SIZE-1:6]	Base address of the region, excluding the lower 6 bits. The base address must be an integer multiple of the region size, as programmed in the Size field of the corresponding MPU_RDP descriptor. 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.

13.1.10 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_RDP1: 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 13-14 MPU Region Descriptor Permissions Registers

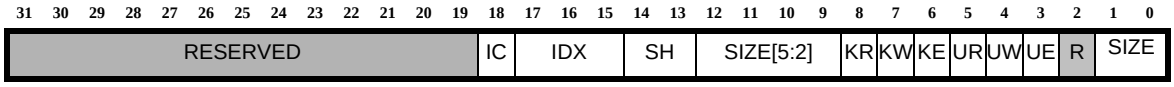


Table 13-9 MPU Region Descriptor Permission Registers Field Description

Field	Bit	Description
SIZE	[12:9] and [1:0]	Size; The size of the region is a 6-bit field, the MSB bits are represented in [12:9] and the two LSB bits are represented in [1:0]. Together these fields specify the size of the region in bytes: MPU Region Size $=2^{((Size+1))}$ bytes Note: The MPU region size has the following constraints: ■ MPU region size >= minimum cache line size of the L1 cache **Note**: If addr_size is 32 (HS5x), bit 12 is ignored (that is size field is 5 bits). If addr_size is 64, size field is 6 bits.
UE	[3]	General Execute Permission ■ `0`: Instruction fetch is not permitted ■ `1`: Instruction fetch is permitted when in user
UW	[4]	General Write Permission ■ `0`: Region is write-protected ■ `1`: Region is writeable when in user

Table 13-9 MPU Region Descriptor Permission Registers Field Description

Field	Bit	Description
UR	[5]	General Read Permission ■ 0: Region is read-protected. ■ 1: Region is readable when in user 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 ■ 1: Instruction fetch is permitted granted when in kernel mode only Note that user mode execution does not imply kernel-mode write.
KW	[7]	Kernel-only Write Permission ■ 0: Region is write-protected ■ 1: Region is writeable when in kernel mode only Note that user mode write does not imply kernel-mode write.
KR	[8]	Kernel-only Read Permission ■ 0: Region is read-protected. ■ 0: 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. User mode read does not imply kernel-mode read.
SH	[14:13]	Shareability attribute applied to this region: ■ 0x0: Non-shareable. Data in this region can be cached, but is considered private, and hence no snooping occurs to addresses in this region ■ 0x1: Reserved. Treated as inner-shareable ■ 0x2: Outer-shareable. Data in this region is considered to be shareable across the system. The region is included in the global coherency domain, and snooping requests are propagated through the inter-cluster interconnect to maintain global coherency ■ 0x3 Inner-shareable. Data in this region is considered to be shareable within the local cluster. The region is included in the local (or inner) coherency domain, and snooping is limited to this inner domain
IDX	[17:15]	This 3-bit field is a pointer to a field in the memory attribute auxiliary register where the memory attributes are specified. These attributes are applicable to data accesses (LD/ST) and not for Instruction fetch. See MPU_MEM_ATTR register.

Table 13-9 MPU Region Descriptor Permission Registers Field Description

Field	Bit	Description
IC	[18]	Code cacheability for the default MPU region ■ `0x0`: Code is cacheable in all levels of the cluster cache hierarchy ■ `0x1`: Code is not cacheable in any level of the cluster cache hierarchy (L1, L2, and so on) Notes: ■ When this bit is set to `1`, the L1 instruction cache is not automatically flushed. Any code in the uncacheable 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_EN[12]==1).

Caution

A region descriptor must not be enabled unless all fields have been set to well-defined values. For example, the region size must be set to a non-reserved value, and all reserved bits must be zero. You must disable the MPU before any region descriptors are updated by setting the MPU_EN register to 0x00000000. After all of the region descriptors have been set to well-defined values, the MPU can be re-enabled by setting MPU_EN to 0x40000000.

Any MPU region that is programmed with an undefined region size assumes the minimum region size. If the base address is not a multiple of the region size, the effective base address is the highest address that is less than the given address and which is also a multiple of the region size. In other words, the base address is the address of the region containing the programmed address.

Preventing speculative instruction fetch

Read-sensitive memory regions must have both UE (user-mode execution permission) and KE (kernel-mode execution permission) bits set to zero to avoid speculative fetches.

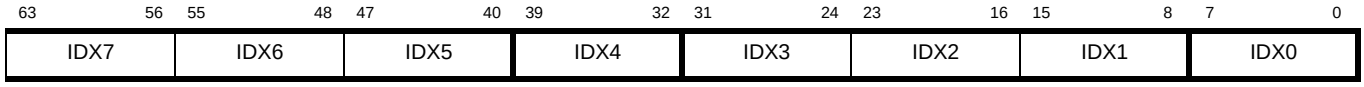
13.1.11 MPU Attribute Register, MPU_MEM_ATTR

Address: 0x442

Access: RW

Reset 0x7979_7979_7979_7979

Figure 13-15 MPU_MEM_ATTR



The memory attributes of a MPU region are not directly stored in the MPU descriptors.

The descriptor only contains a 3-bit pointer to one of the 8 available fields of the MEM_ATTR auxiliary register.

Each field is 8-bit wide and defines volatile or normal memory regions. The top four bits define the outer cache attributes. These are exported on the system bus (AXI/ACE interface). The bottom four bits are attributes for the volatile or normal memory regions.

The B attribute for this region is conveyed on the AWCACHE[0] bufferable attribute.

Note the AXI bufferable and the early-write-buffer-acknowledge attributes are configured independently. This allows non-posted writes to be buffered in the processor write buffer.

Volatile Memory Region

A volatile memory region is defined by programming bit[0] to 0.

Table 13-10 Volatile Memory Region

MEM_ATTR[7:4]	MEM_ATTR[3:0]	Description
000B	x000	Volatile, no early write acknowledge, strict ordering. See Write Buffer for more information.
000B	x010	Volatile, early write acknowledge, strict ordering. See Write Buffer for more information.
000B	x100	Volatile, no early write acknowledge, relaxed ordering. See Write Buffer for more information.
000B	x110	Volatile, early write acknowledge, relaxed ordering. See Write Buffer for more information.
Others: Reserved, treated as 000B	x: Reserved, treated as 0	

Normal Region

A normal memory region is defined by programming bit[0] to 1.

Table 13-11 Normal Memory Attributes

MEM_ATTR[7:4]	MEM_ATTR[3:0]	Description
000B	0xx1	Normal memory uncached
0W11	1xx1	Normal memory inner-write back
Others: Reserved, treated as 000B	x: Reserved, treated as 0	

The W and B are outer attributes reflected on the AxCACHE AXI4/ACE system bus.

Table 13-12 AXI4 Attributes

Memory Type	AxCACHE[3]	AxCACHE[2]	AxCACHE[1] Modifiable	AxCACHE[0] Bufferable
Volatile memory	0	0	0	B
Normal memory uncached	0	0	1	B
Normal memory, inner write-back cacheable	W	1	1	1

13.1.12 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 13-16 MPU_BUILD Register

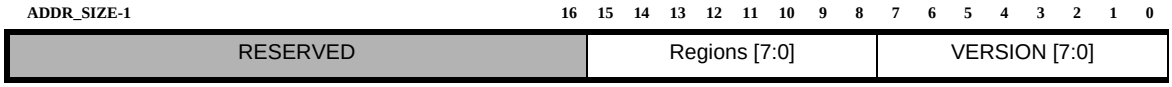


Table 13-13 MPU_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version ■ 0x20: ARCv3 implementation 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 for16 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.

13.1.13 MPU Exceptions

See Table 7-3 on page 204.

14 Protection Schemes

ARCv3 ISA-based processors provide several memory-protection options, which are described in this chapter. See the Data Book for your specific processor to determine which options are available.The protection options include stack protection, that checks overflow and underflow of reserved stack space.

- Stack protection: checks overflow and underflow of reserved stack space

The different protection schemes may be combined to achieve several levels of protection against malicious or misbehaving code in critical applications.

14.1 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 run time, 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.

14.1.1 Build Configuration Register

Stack Region Configuration Register, STACK_REGION_BUILD (0xC5) identifies the presence and version of stack checking in the build.

14.1.2 Enabling Stack Checking

The kernel mode 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

14.1.3 Specifying Stack Regions

Specify the memory regions reserved for the stack by loading their addresses into the following auxiliary registers. Separate register pairs define the kernel-mode and user-mode stack regions.

The stack grows downward from higher memory address to a lower memory address.

- `USTACK_TOP` (0x260) is the user-mode lower address limit.
- `USTACK_BASE` (0x261) is the user-mode upper address limit.
- `KSTACK_TOP` (0x264) is the kernel-mode lower address limit.
- `KSTACK_BASE` (0x265) is the kernel-mode upper address limit.

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.

14.1.4 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.

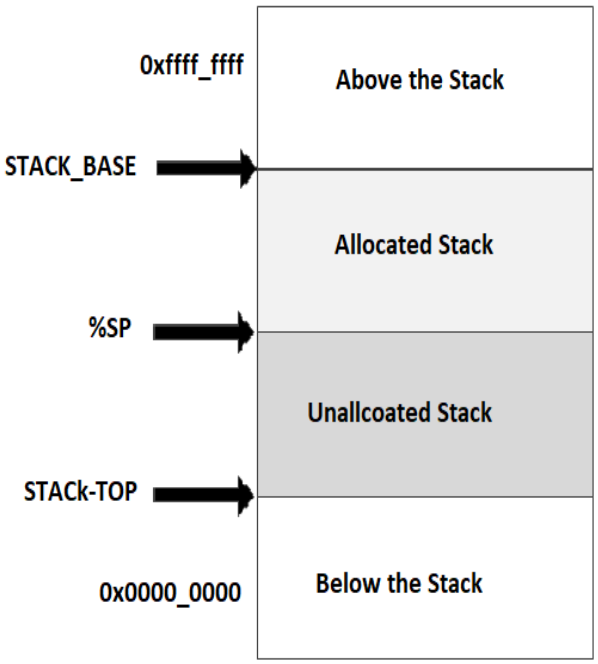
14.1.4.1 Stack Regions

Figure 14-1 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 ARCv3-based processors.
- The allocated stack region is defined as all addresses from (and including) %SP up to (but excluding) STACK_BASE.
- The unallocated stack region is defined as all addresses from (and including) STACK_TOP up to (but excluding) %SP.
- STACK_BASE points to the first 32-bit word above the reserved stack space, and hence all addresses greater than or equal to STACK_BASE are above the stack.
- STACK_TOP points to the last allocatable word in the stack region, and hence all addresses less than STACK_TOP are below the stack.

Note The stack grows downward from higher memory address to a lower memory address.

Figure 14-1 Stack regions



14.1.4.2 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 14-2 and Figure 14-2 illustrate the conditions checked in three possible scenarios.

Figure 14-2 Stack Checking Scenarios

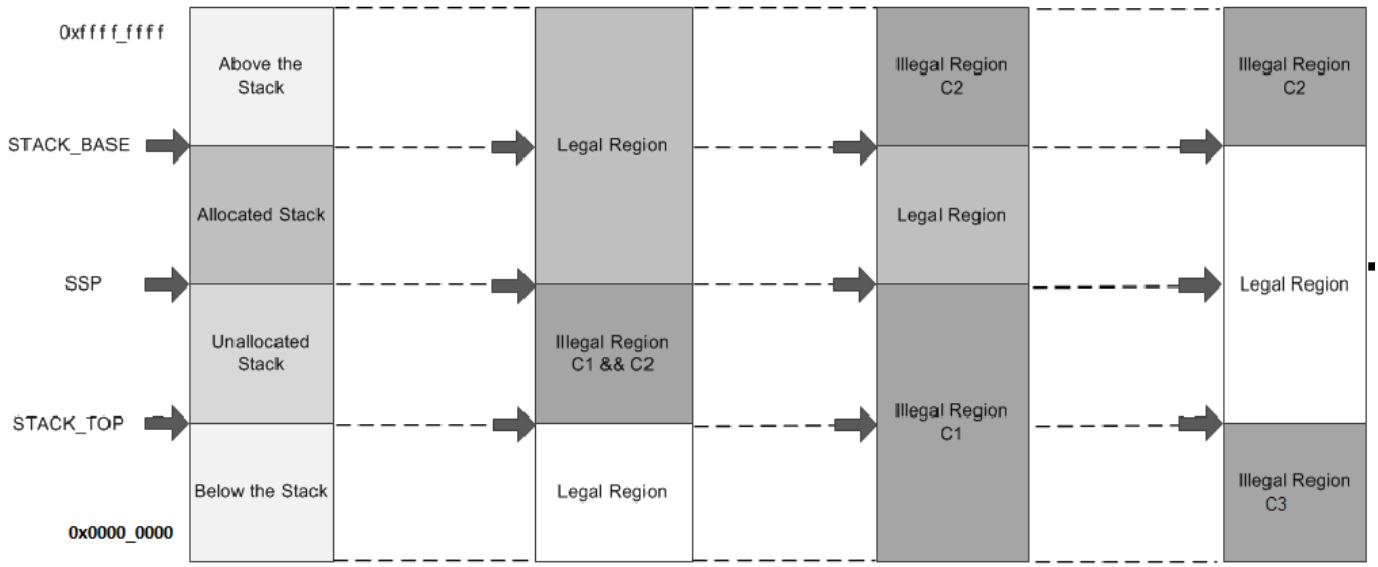


Table 14-1 Stack Checking Specification

Conditions	Scenario 1 Non-%SP-Based Address. **Example:** `LD r0, [r1];`	Scenario 2 %SP-Based, No Auto-Update of %SP **Example:** `LD r0, [r28];`	Scenario 3 %SP-Based, with Auto-Update of %SP **Example:** `LD.a r0, [r28];`
Condition C1	address < SSP	address < SSP	
Condition C2	address ≥ STACK_TOP	address ≥ STACK_BASE	address ≥ STACK_BASE
Condition C3		address < STACK_TOP	address < STACK_TOP
Error Condition	C1 && C2	C1 || C2 || C3	C2 || C3

14.1.5 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 7-3.

14.1.6 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.

14.1.7 Stack Region Auxiliary Register Set

The four auxiliary registers to support Stack Checking are mentioned in Table 14-2. The registers contain memory address fields that needs to be widened to work with 64-bit addresses.

Table 14-2 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

14.1.7.1 User Stack Region Top Address, USTACK_TOP

USTACK_TOP Address:	0x260
Access	RW
Reset	0x0000_0000

You can use the stack region top address register, USTACK_TOP to set the top address of the valid stack region in memory. This auxiliary register is present in the processor if you have configured the processor to include stack checking.

Figure 14-3 USTACK_TOP

All user-mode 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 user-mode 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 user-mode 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.

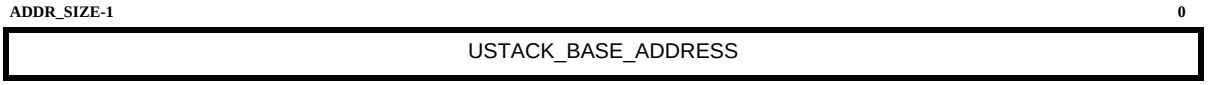
14.1.7.2 User Stack Region Base Address, USTACK_BASE

USTACK_BASE Address:	0x261
Access	RW
Reset	0x0000_0000

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. 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.

Figure 14-4 USTACK_BASE



All user-mode 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 user-mode 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 user-mode 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.

14.1.7.3 Kernel Stack Region Top Address, KSTACK_TOP

KSTACK_TOP Address:	0x264
Access	RW
Reset	0x0000_0000

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. This auxiliary register is present in the processor if you have configured the processor to include stack checking.

Figure 14-5 KSTACK_TOP

All kernel-mode 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 kernel-mode 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 kernel-mode 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.

14.1.7.4 Kernel Stack Region Base Address, KSTACK_BASE

KSTACK_BASE Address:	0x265
Access	RW
Reset	0x0000_0000

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. 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.

Figure 14-6 KSTACK_BASE

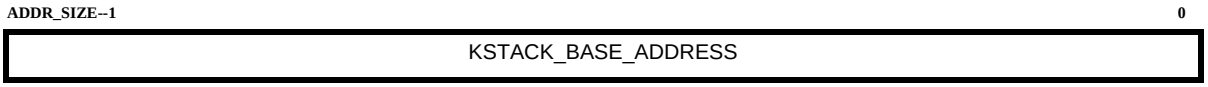
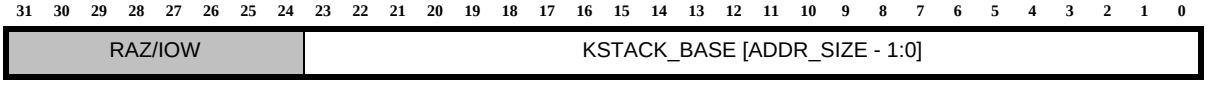


Figure 14-7 KSTACK_BASE when ADDR_SIZE < 32



All kernel-mode 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 kernel-mode 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 kernel-mode 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.

14.1.7.5 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 14-8 STACK_REGION_BUILD Register

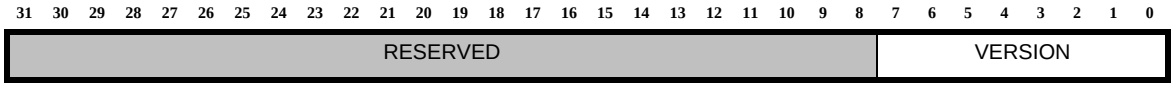


Table 14-3 STACK_REGION_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	**Version of stack region checking** ■ `0x01`: ARC 700 style stack checking ■ `0x02`: ARCv3 style stack checking

14.1.8 Stack Checking Exceptions

See Table 7-3.

14.2 Coexisting Protection Schemes

Stack Checking can coexist with 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:
  - Stack Checking violations set bit 1
  - MPU permission violations set bit 2
  - MMU permission violations set bit 3

    Hence, multiple protection violations can be reported by raising a single exception. See the Table 7-3 for more information about the exception information for each of the protection schemes.

15 Memory Management Unit

15.1 MMU Introduction

This section describes the ARCv3 Memory Management Unit (MMU).

The Memory Management Unit (MMU) enables a system to run multiple tasks as independent programs running in their own virtual address space.

The MMU uses a set of memory mapped translation tables to control address translation, access permissions, and to determine the memory attributes of memory access performed by the processor.

The main features of the ARCv3 MMU are:

- Pre-defined translation table format
- Hardware page table walk – hardware managed TLBs

15.2 High-Level Architecture

The ARCv3 MMU supports the following address spaces:

- MMUv48: Virtual address space up to 48 bits and physical address space of 48 bits (ARC64)
- MMUv52: Virtual address space up to 52 bits and physical address space of 52 bits (ARC64)

15.2.1 MMUv48

MMUv48 is a page-based virtual memory system with 48-bit virtual address and 48-bit physical address. Virtual addresses are resolved at a minimum granularity of either 4KB, 16KB or 64KB. The minimum granularity is a fixed build-time parameter. The minimum granularity defines the minimum page size supported by the MMU. Moreover, it also defines the size of the page tables and the maximum number of table lookups required to resolve a virtual address. Each translation table entry is always 64-bit wide in MMUv48.

Table 15-1 MMUv48 properties

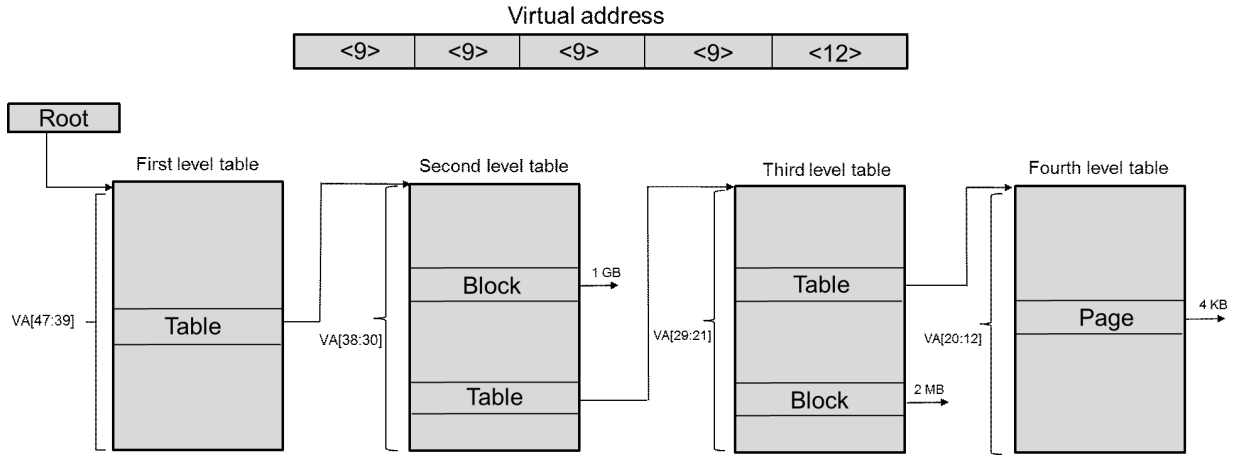
Property	4KB	16KB	64KB
Max number of entries in a translation table	512	2048	8192
Max lookup levels	4	4	3

Table 15-1 MMUv48 properties

Address bits resolved per table lookup	9	11	13
Untranslated bits	[11:0]	[13:0]	[15:0]

Table 15-1 shows an example of page table walk with a 4KB minimum granularity. The virtual address is partitioned in 5 fields. The first four fields (from left to right) are used to index their respective lookup tables. The last field represents the page offset and it is never translated.

Figure 15-1 MMUv48 Page Table Walk with 4KB Minimum Granularity



The page table walk for the MMUv48-16KB and MMUv48-64KB are depicted below.

Figure 15-2 MMUv48 Page Table Walk with 16KB Minimum Granularity

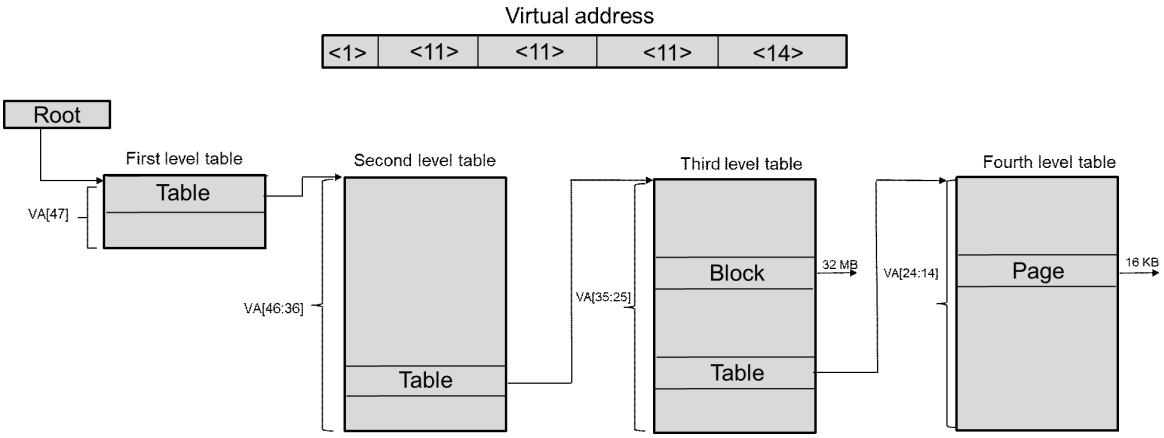
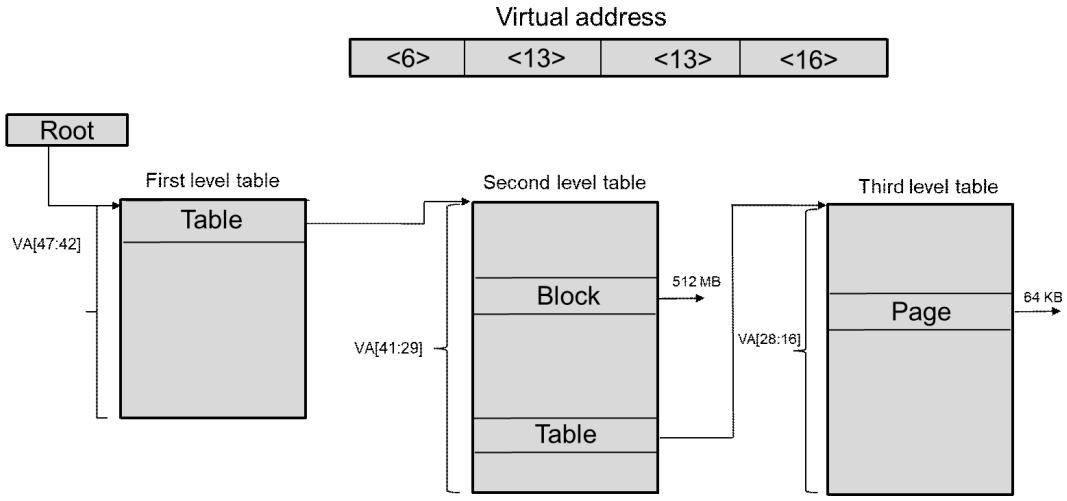


Figure 15-3 MMUv48 Page Table Walk with 64KB Minimum Granularity

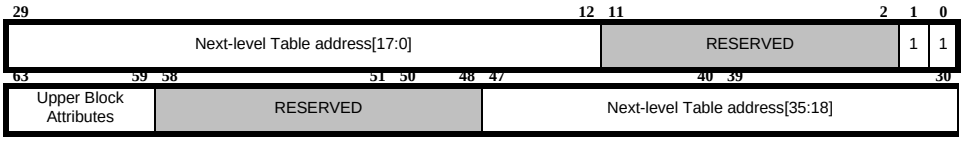


15.2.1.1 MMUv48 Translation Table Format

The format of the page descriptor depends on the level of the table where the descriptor is stored. The first level always contains a pointer to the next level, i.e.: it is always a table descriptor (never a block descriptor). The second level may contain a pointer to the next level of the table hierarchy or define a 1GB block. The third level may contain a pointer to the next level of the table hierarchy or define a 2MB block. The fourth level descriptor always defines a memory page.

15.2.1.1.1 Table Descriptors

Figure 15-4 MMUv48 Level 1 Block Descriptor

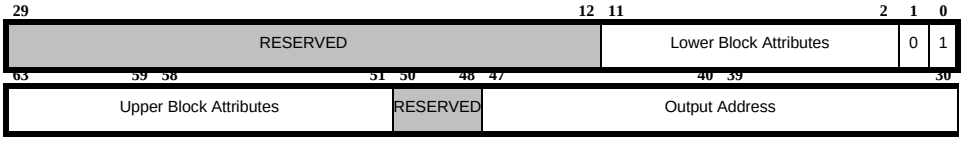


Fields:

- **Bit[0]:** 1 indicates a valid descriptor
- **Bit[1]:** 1 indicates a table descriptor
- **Next level table address:** Pointer to the next level table.
  - **MMUv48-4K**:The address of the next level table is 4KB aligned and therefore bits [11:0] are zero
  - **MMUv48-16K**:The address of the next level table is 16KB aligned and therefore bits [13:0] are zero
  - **MMUv48-64K**:The address of the next level table is 64KB aligned and therefore bits [15:0] are zero
- **Table attributes:** Protection attributes.

15.2.1.1.2 Second level Block Descriptors

Figure 15-5 MMUv48 Second level block descriptor



Fields:

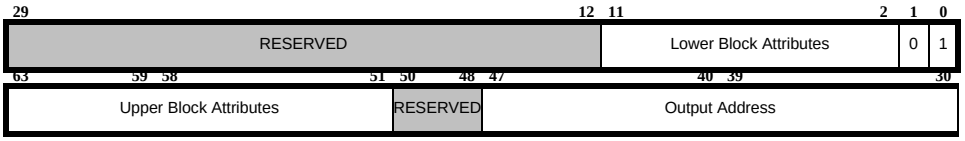
- **Lower block attributes:** Protection and memory attributes for the 1GB block of memory (MMUv48-4KB) or 512MB (MMUv48-64KB).

Output address:

- **MMUv48-4K:** Bits[47:30] of the 1GB aligned physical address
- **MMUv48-16K:** N/A. Only table descriptors allowed at this level
- **MMUv48-64K:** Bits[47:29] of the 512MB aligned physical address
  - **Upper block attributes:** Protection attributes for the block of memory.

15.2.1.1.3 Third level Block Descriptors

Figure 15-6 MMUv48 Third level block descriptor



Fields:

- **Lower block attributes:** Protection and memory attributes for the 2MB (MMUv48-4K) or 32MB (MMUv48-16KB) block of memory.

Output address:

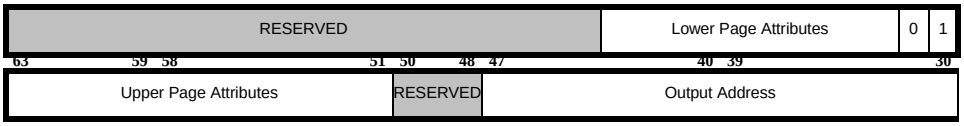
- **MMUv48-4K:** Bits[47:21] of the 48-bit 2MB aligned physical address
- **MMUv48-16K:** Bits[47:25] of the 48-bit 32MB aligned physical address
- MMUv48-64K:
- Bits[47:16] of the 48-bit 64KB aligned physical address
  - **Upper block attributes:** Protection attributes for the block of memory.

15.2.1.1.4 Third Level Descriptor Format

This is only applicable to the MMUv48-64K configuration. The only valid format at this level is a page descriptor. Therefore, the only valid value of bits[1:0] of this descriptor is 11.

Figure 15-7 MMUv48-64K fourth level page descriptor

Figure 15-7 MMUv48-64K fourth level page descriptor



Fields:

- **Lower page attributes:** Protection and memory attributes for the 64KB page of memory.

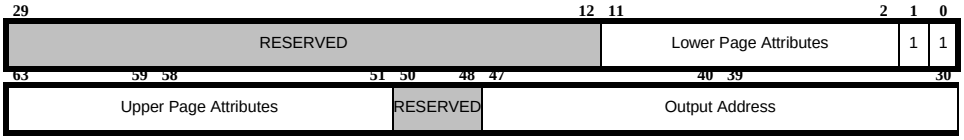
Output address:

- **MMUv48-4K:** N/A
- **MMUv48-16K:** N/A
- MMUv48-64K:
- Bits[47:16] of the 48-bit 64KB aligned physical address
  - **Upper block attributes:** Protection attributes for the page of memory.

15.2.1.1.5 Fourth level descriptor format

The only valid format at this level is a page descriptor. Therefore, the only valid value of bits[1:0] of this descriptor is 11.

Figure 15-8 MMUv48 fourth level page descriptor



Fields:

- **Lower page attributes:** Protection and memory attributes for the 4KB page of memory.

Output address:

- **MMUv48-4K:** Bits[47:12] of the 48-bit 4KB aligned physical address
- **MMUv48-16K:** Bits[47:14] of the 48-bit 16KB aligned physical address
- MMUv48-64K:
- N/A
  - **Upper block attributes:** Protection attributes for the page of memory.

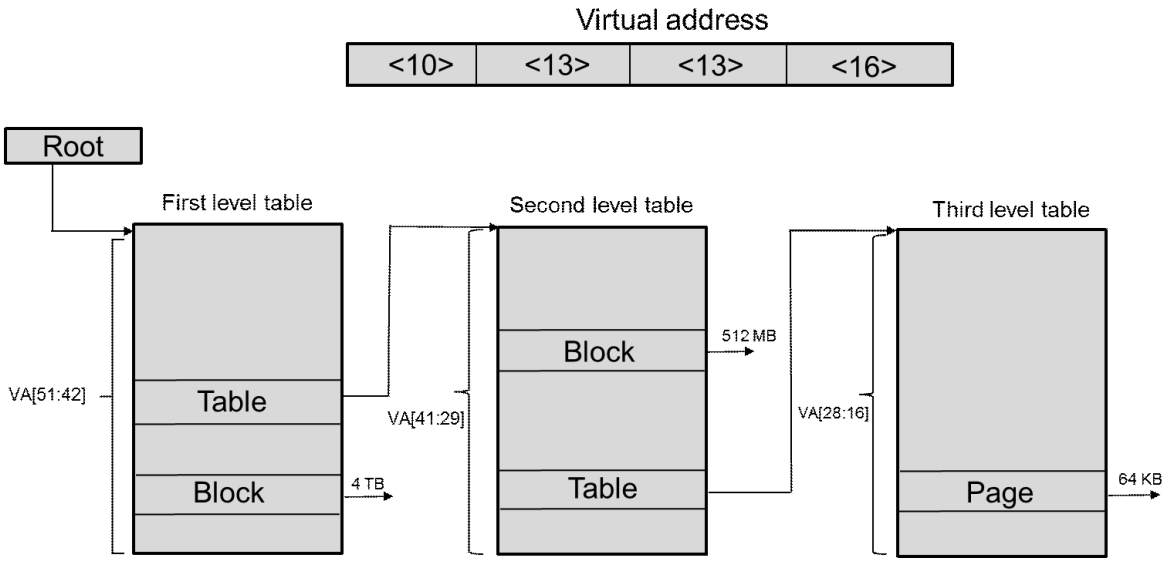
15.2.2 MMUv52

MMUv52 is a page-based virtual memory system with 52-bit virtual address and 52-bit physical address. Virtual addresses are resolved at a granularity of 64KB. Each translation table entry is always 64-bit wide in MMUv52.

Property	64KB
Max number of entries in a translation table	8192
Max lookup levels	3
Address bits resolved per table lookup	13
Untranslated bits	[15:0]

Following figure shows an example of page table walk. The virtual address is partitioned in 5 fields. The first four fields (from left to tight) are used to index their respective lookup tables. The last field represent the page offset and it is never translated.

Figure 15-9 MMUv52 Page Table Walk Overview



15.2.2.1 MMUv52 Translation Table Format

The page descriptor format depends on the level of the descriptor. A table descriptor is permitted on the first level and second level of table lookup. A table descriptor on the third level is not permitted. A block descriptor is permitted on the first (4TB) and second level (512MB). The third level always point to a 64KB page.

15.2.2.1.1 Table Descriptors

Figure 15-10

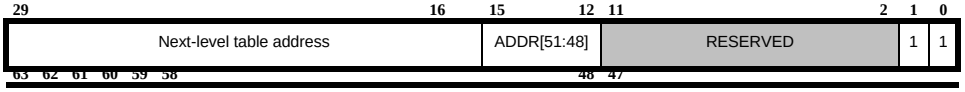


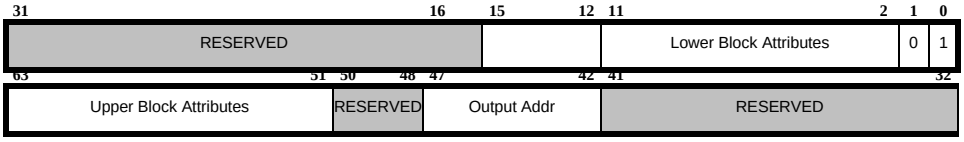
Figure 15-10

Fields:

- **Bit[0]:** 1 indicates a valid descriptor
- Bit[1]:
- 1 indicates a table descriptor
- Next level table addr:
- The 52-bit address to the next level table is stored as follows:
  - **Addr[51:48]:** stored at position[15:12]
  - Addr[47:16]:
  - stored at position[47:16]
- **Table attributes:** Protection attributes.

15.2.2.1.2 First-level Block

Figure 15-11



Fields:

- **Lower block attributes:** Protection and memory attributes for the 4TB memory block.

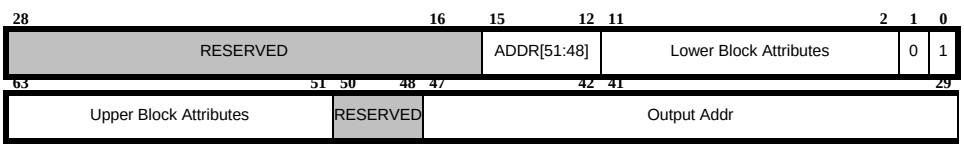
Output block address:

The 52-bit address of this 4TB memory block is stored as follows:

- **Addr[51:48]:** stored at position[15:12]
- Addr[47:42]:
- stored at position[47:42]
  - **Upper block attributes:** Protection attributes for the 4TB memory block.

15.2.2.1.3 Second level block descriptor

Figure 15-12



The only difference between the second and first level descriptor is the size of the memory block.

Fields:

- **Lower block attributes:** Protection and memory attributes for the 512 MB memory block.

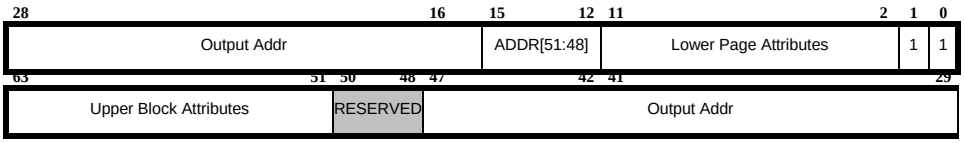
Output block address:

The 52-bit address of this 512 MB memory block is stored as follows:

- **Addr[51:48]:** stored at position[15:12]
- Addr[47:29]:
- stored at position[47:29]
  - **Upper block attributes:** Protection attributes for the 512 MB memory block.

15.2.2.1.4 Third level Descriptor Format

Figure 15-13



The only valid format at this level is a 64KB page descriptor.

Fields:

- **Bit[0]:** 1 indicates a valid descriptor
- **Bit[1]:** 1 indicates a page descriptor
- **Lower page attributes:** Protection and memory attributes for the 64KB memory page.

Output address: The 52-bit page address is stored as follows:

- **Addr[51:48]:** stored at position[15:12]
- Addr[47:16]: stored at position[47:16] Upper page attributes: Protection attributes for the 64KB memory page.

15.2.3 Table, Block and Page Attributes

This section describes the table, block, and page attributes stored in the descriptors.

15.2.3.1 Lower Block/Page Attributes

The lower attributes of a block or page is defined as follows:

Figure 15-14 Block and Page Lower Attributes

11	10	9	8	7	6	5	4	3	2
nG	AF	Shareability		Access Permission		Res	Memory attr index		

Fields:

- **Memory attribute index, mem_attr_idx:** This 3-bit field is a pointer to a field in the memory attribute auxiliary register where the memory attributes are specified.
- Access permission, AP: Controls data access permissions to the block or page of memory.
  - Bit[6]:
    - 0: indicates only kernel mode access is permitted
    - 1: indicates user or kernel mode accesses are permitted
  - Bit[7]:
    - 0: Read and write accesses are permitted
    - 1: Read-only access is permitted

The following summarizes the semantics of the access permission bits:

Table 15-2 Access Permission Attributes

AP	Access
2'b00	Read-write access in kernel mode only
2'b01	Read-write access in user mode and read-write access in kernel mode if MMU_CTRL.KU is set
2'b10	Read-only access in kernel mode only
2'b11	Read-only access in user mode and read-only access in kernel mode if MMU_CTRL.KU is set

- **Shareability, SH:** The shareability attributes is only applicable to normal memory regions and are ignored for volatile memory regions. Furthermore, the shareability fields is only applicable to a region that is not marked as non-cacheable by the field pointed to by the mem_attr_idx field.
  - **00: Non-shareable**. Data in this region can be cached, but is considered **private**, and hence no snooping occurs to addresses in this region
  - 01: Reserved
  - 10: **Outer-shareable**. Data in this region is **shareable across the system**. The region is included in the global coherency domain, and snooping requests are propagated through the inter-cluster interconnect to maintain global coherency
  - 11: **Inner-shareable.** Data in this region is **shareable within the local cluster**. The region is included in the local (or inner) coherency domain, and snooping is limited to this inner domain

Table 15-3 Shareability Attributes

SH	Normal memory

Table 15-3 Shareability Attributes

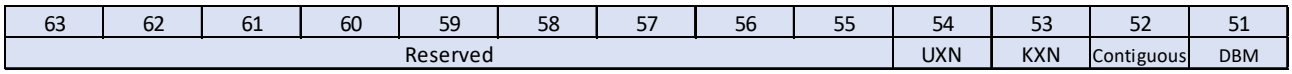
2'b00	Non-shareable
2'b01	Reserved
2'b10	Outer-shareable
2'b11	Inner-shareable

- **Access Flag, AF:** The Access flag indicates when a block or page of memory is accessed for the first time since the Access flag is cleared. This flag is managed by software in the ARCv3 MMU
- **Non-global, nG**:
  - **0:** Global translation, available for all processes
  - 1: Translation is non-global, or process specific and relates to the current ASID

15.2.3.2 Upper Block/Page Attributes

The upper block/page attributes are defined as follows:

Figure 15-15 Block and Page Upper Attributes



Fields:

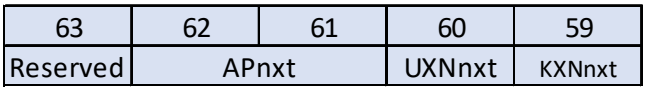
- **Dirty bit modifier, DBM:** This bit indicates a page or block of memory is modified. The dirty bit modifier is managed by software in the ARCv3 MMU
- **Contiguous, CTG:** This bit provides a hint to TLB that allows a single TLB entry to cover 16 translation table entries. This bit is asserted to indicate 16 translation table entries with the same protection and memory attributes map to a contiguous address range. Moreover, these 16 table entries must be on the same minimum granularity translation node. To use the contiguous bit on the last level table, bits[20:16] of the 16 4KB pages must be the same. This bit increases the TLB coverage, allowing one single TLB entry to cover 16*4KB pages. The continuous bit hint is only considered by the HW when the minimum granule is 4KB and it is only applicable for the last level (leaf) of the page table
- **Kernel-execute-never, KXN:** This bit is asserted to indicate execution is not permitted in kernel-mode. The KXN bit also applies to kernel mode speculative instruction fetch
- **User-Execute-never, UXN:** This bit is asserted to indicate execution is not permitted in user-mode. The UXN bit also applies to user mode speculative instruction fetch

15.2.3.3 Table attributes

> **Note** Software defining the translation table must mark any read-sensitive memory region as UXN and KXN to avoid any kind of speculative instruction fetch from these regions.

The table attributes for the first and second level table descriptors are defined as follows:

Figure 15-16 Table Attributes



The table attributes are applicable to the next levels of lookup. The intention of the table attributes is to restrict instruction fetch and access permissions at subsequent levels of lookup.

Fields:

- Kernel-execute-never next, KXNnxt:
  - **0:** This bit has no effect
  - 1: The Kernel-Execute-Never bit (KXN) shall be treated as 1 on all subsequent lookup levels, regardless of the actual value of the KXN field
- User-Execute-never next, UXNnxt:
  - **0:** This bit has no effect
  - 1: The User-Execute-Never bit (UXN) shall be treated as 1 on all subsequent lookup levels, regardless of the actual value of the UXN field

Access permission next, APnxt: Restricts permissions to subsequent lookup levels, according to the following table:

Table 15-4 APnxt Permissions

APnxt	Access
2'b00	No effect on permission
2'b01	User mode accesses are not permitted
2'b10	Write accesses are not permitted
2'b11	Write accesses are not permitted, read accesses not permitted in user mode

> **Note** A change in APnxt attributes requires a coarse-grain invalidation of TLB entries, as the effect of these attributes may be held in many TLB entries.

15.2.3.4 Memory Region Attributes

Another function of the MMU is to determine the memory attributes of a memory access performed by the processor. The memory attributes control the memory type (normal or volatile), cacheability and shareability of memory page.

The ARCv3 MMU determines memory attributes indirectly. The descriptors contain a 3-bit pointer (mem_attr_idx) to fields in auxiliary registers and the memory attributes are determined by these fields.

Each field is 8-bit wide and contain attributes defining volatile or normal memory regions. The top 4 bits of each field indicate either a volatile or a normal memory region. The bottom 4 bits of each field further define attributes of either the volatile or the normal memory region.

Volatile region:

A volatile region is never cached in any cache inside the ARC cluster. Furthermore, loads and stores targeting the volatile region are not optimized by the core: writes are never merged and store to load forwarding is not permitted.

Normal memory region:

The following attributes are available for a normal memory region:

- **Inner cacheability:** This indicates whether the memory region can be cacheable in the L1 or L2 cache
- **Outer memory attributes:** These memory attributes are applicable to caches outside the ARC cluster. They are conveyed by AXI/ACE-Lite system bus protocol as hints

The following attributes are available for a volatile memory region:

- **Early-write-acknowledgment**: Indicates a write may receive an early write response from the write buffer. The early-write-acknowledgment can be programmed independently of the bufferable attribute conveyed on the AXI bus (AxCACHE[0]). This means that posted or non-posted writes may receive an early-write-acknowledgment from the write buffer.
- **Relaxed or Strict ordering:** Reads are executed in order with respect to other reads. Writes are executed in order with respect to other writes. The relaxed ordering attribute relies on the external memory or peripheral subsystem to not reorder reads and writes. The strict ordering attribute serializes read and write transactions and therefore the ordering of volatile transactions is guaranteed independently of the memory or peripheral subsystem.

15.2.3.5 Instruction Execution And Data Access Permissions

The instruction execution permission settings in the ARCv3 MMU is controlled by the UXN and KXN bits. Furthermore, the MMU_CTRL.WX (Write-Executable) bit prevents that writable pages at a given privilege level (user or kernel) have execution permissions at that and higher privilege levels. As an example, when the MMU_CTRL.WX = 0, a page that is writable in user mode does not have execution permissions at user or kernel mode.

The settings of both UXN and KXN bits also prevent speculative instruction fetch from that memory region.

The data access permissions in the ARCv3 MMU are controlled by the AP/APnxt (access permission) bits as well as the MMU_CTRL.KU bit.

Table 15-5 shows all possible scenarios.

Table 15-5 Instruction Execution and Data Access Permissions

UXN	KXN	AP	U-mode Execution	K-mode Execution	Speculative Fetch	U-mode Read (LD)	U-mode Write (ST)	K-mode Read (LD)	K-mode Write (ST)
0	0	00	Allowed	Allowed if MMU_CTRL.WX=1	Allowed	Not allowed	Not allowed	Allowed	Allowed
		01	Allowed if MMU_CTRL.WX=1	Allowed if MMU_CTRL.WX=1	Allowed	Allowed	Allowed	Allowed if MMU_CTRL.KU=1	Allowed if MMU_CTRL.KU=1
		10	Allowed	Allowed	Allowed	Not allowed	Not allowed	Allowed	Not allowed
		11	Allowed	Allowed	Allowed	Allowed	Not allowed	Allowed if MMU_CTRL.KU=1	Not allowed
0	1	00	Allowed	Not allowed	Allowed	Not allowed	Not allowed	Allowed	Allowed
		01	Allowed if MMU_CTRL.WX=1	Not allowed	Allowed	Allowed	Allowed	Allowed if MMU_CTRL.KU=1	Allowed if MMU_CTRL.KU=1
		10	Allowed	Not allowed	Allowed	Not allowed	Not allowed	Allowed	Not allowed
		11	Allowed	Not allowed	Allowed	Allowed	Not allowed	Allowed if MMU_CTRL.KU=1	Not allowed
1	0	00	Not allowed	Allowed if MMU_CTRL.WX=1	Allowed	Not allowed	Not allowed	Allowed	Allowed
		01	Not allowed	Allowed if MMU_CTRL.WX=1	Allowed	Allowed	Allowed	Allowed if MMU_CTRL.KU=1	Allowed if MMU_CTRL.KU=1
		10	Not allowed	Allowed	Allowed	Not allowed	Not allowed	Allowed	Not allowed
		11	Not allowed	Allowed	Allowed	Allowed	Not allowed	Allowed if MMU_CTRL.KU=1	Not allowed
1	1	00	Not allowed	Not allowed	Not allowed	Not allowed	Not allowed	Allowed	Allowed
		01	Not allowed	Not allowed	Not allowed	Allowed	Allowed	Allowed if MMU_CTRL.KU=1	Allowed if MMU_CTRL.KU=1
		10	Not allowed	Not allowed	Not allowed	Not allowed	Not allowed	Allowed	Not allowed
		11	Not allowed	Not allowed	Not allowed	Allowed	Not allowed	Allowed if MMU_CTRL.KU=1	Not allowed

15.3 Programming Model

15.3.1 Address Spaces

The MMUv48 defines a **virtual address** space up to 48 bits. The **physical address** space defined by MMUv48 is 48 bits. The MMUv48 page table format gives access to the 48-bit physical address at a granularity of 4KB, 16KB, or 64KB.

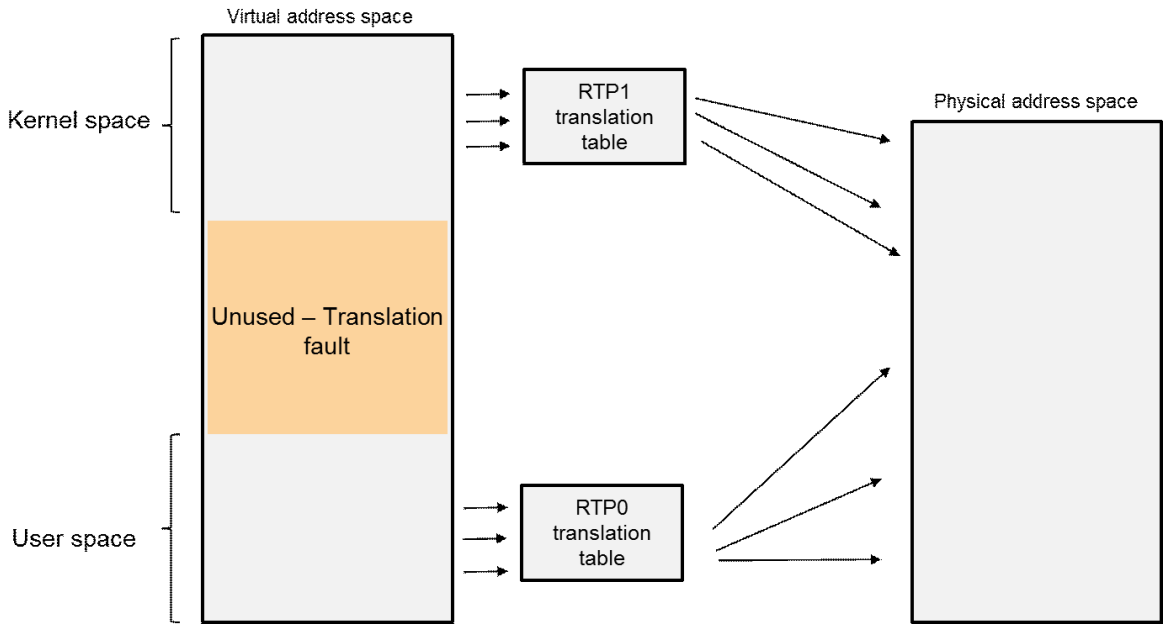
The MMUv52 defines a **virtual address** space up to 52 bits. The **physical address** space defined by MMUv52 is 52 bits. The MMUv52 page table format gives access to the 52-bit physical address at a granularity of 64KB.

15.3.1.1 Root Translation Pointers

The virtual to physical translation is defined by translation tables. The pointer to the root of the translation tables are defined by the Root Translation Pointers auxiliary registers: MMU_RTP0 and MMU_RTP1.

The ARCv3 MMU provides a mechanism whereby the operating system can separate the kernel and user space translation tables. The kernel translation tables rarely change on a context switch, whereas user processes may have its own translation table, requiring a new user space table on a context switch.

Figure 15-17 Two Translation Tables



The size of the virtual region mapped by either MMU_RTP0 or MMU_RTP1 is specified in the Translation Base Control auxiliary register: MMU_TTBC.

15.4 Translation Lookaside Buffer (TLB)

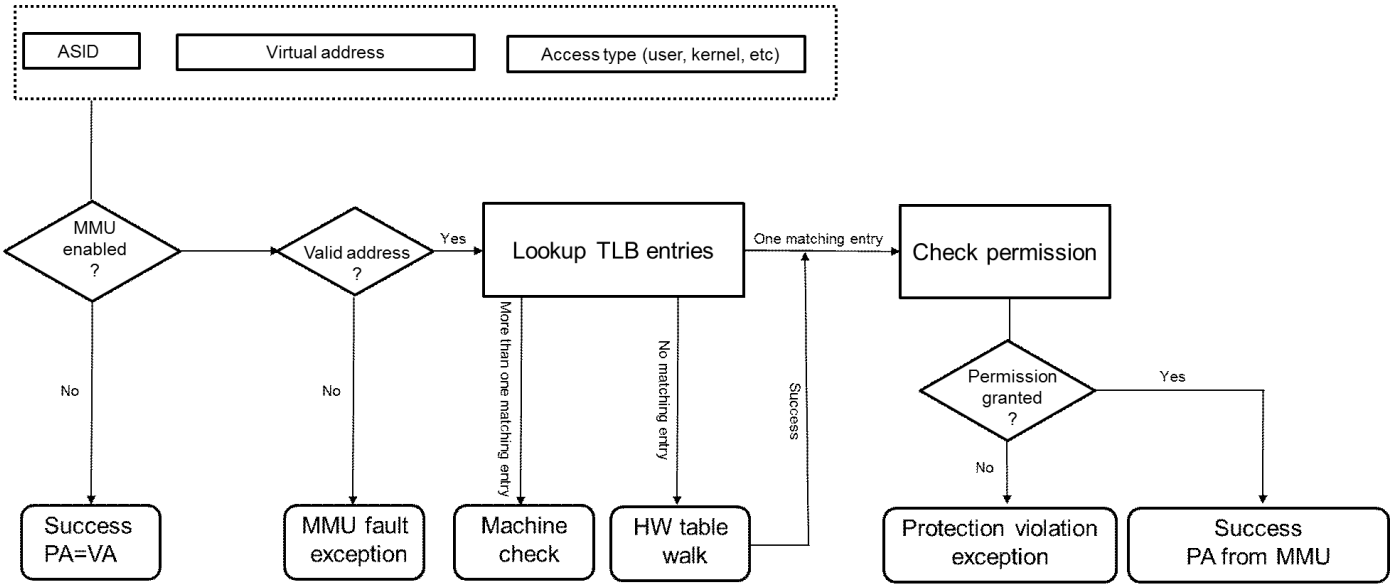
The TLB is a cache for recently accessed translation table entries. The TLB caches virtual and physical addresses, ASID, access permissions and memory attributes.

Each memory reference is first checked against the TLB. In case of a TLB miss, the hardware initiates a process known as a page table walk. The page table walk reads the page tables in memory and possibly cache these entries in the TLB for future accesses.

15.4.1 TLB Lookup Flowchart

The TLB depicted in this flowchart represents the last level TLB. For more information, see *ARCv3-based processor Databook*.

Figure 15-18 TLB Lookup Flowchart



15.4.1.1 TLB Maintenance

When the page tables are modified in memory, the entries that are cached in the TLB may become out-of-sync (stale) with respect to the memory page table. It is responsibility of software (usually the operating system) to invalidate stale entries in the TLB. The micro-TLBs are hardware micro-architecture optimizations and the hardware ensures they are in sync with the last level TLB.

The following TLB maintenance operations are supported:

- Invalidate all TLB entries
- Invalidate TLB entries
  - By virtual address
  - By ASID
  - By virtual address and ASID

The semantics for the TLB maintenance operations are described in the MMU TLB Command Register, MMU_TLB_CMD.

15.4.1.2 TLB Coherency

There are two options to deal with TLB maintenance operations on a multi-core coherent system:

- Inter-processor interrupt (IPI) – software
- Distributed Virtual Memory (DVM) messages – hardware

In the IPI approach, the TLB invalidation is performed locally on one core and it relies on software to generate an IPI interrupt. Upon receiving this interrupt, the other cores perform similar invalidation on their TLBs.

The ARCv3 processors rely on Inter-processor interrupts (IPI) to maintain TLB coherency on a multi-core system.

15.5 Hardware Page Table Walk

Control

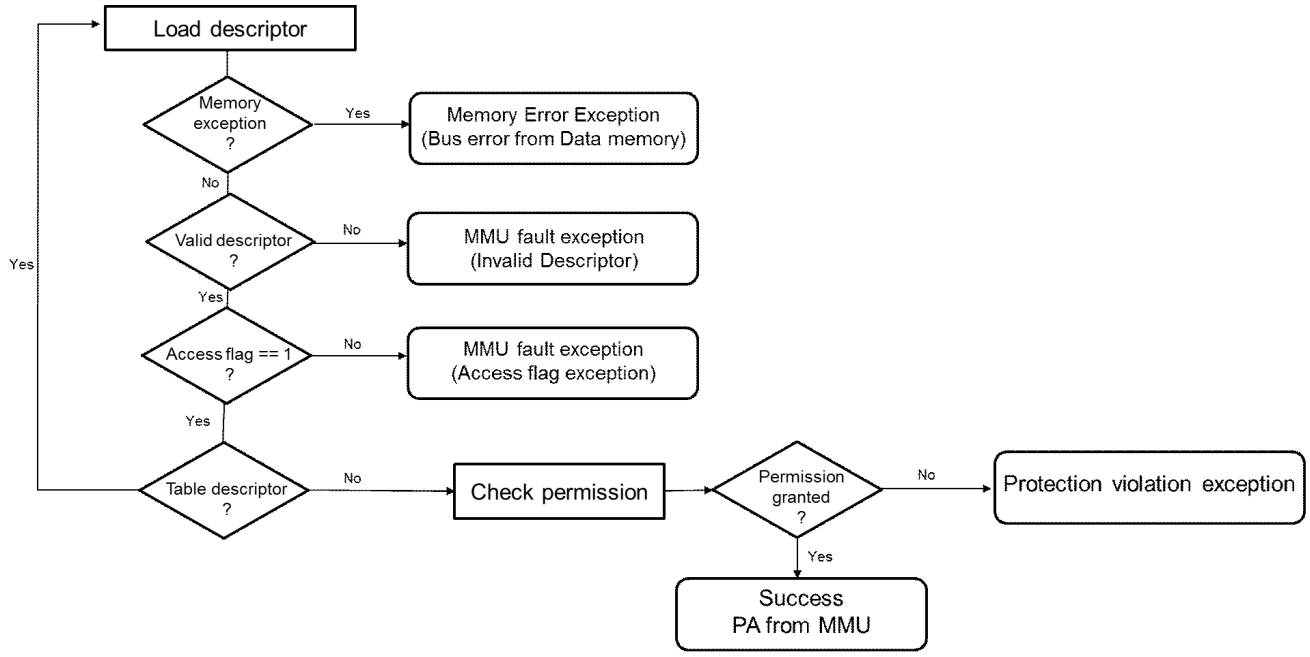
A hardware page table walk is triggered by a miss on the last-level TLB. The physical address of the root of the page table is fetched from either MMU_RTP0 or MMU_RTP1, as described in Root Translation Pointers.

Shareability and cacheability or translation tables

The Translation table base control auxiliary register controls the cacheability and shareability attributes of memory references performed on behalf of page table walks. The page tables could be either cached or uncached.

Information returned by a translation table walk

Figure 15-19 Hardware Page Table Walk Flowchart



The page table walk process describing a virtual to physical translation is illustrated by the following algorithm/pseudo-code. This pseudo-code assumes MMUv48 with 4 lookup levels

Figure 15-20 Page Table Walk Algorithm

1. Let root = MMU_RTP [47:12] and total levels = 4. Let *i = 0*
2. Let pte be the memory value at address root + VPN[i], where VPN[i] is 9 bits wide
3. If accessing the memory location raises a memory error exception, stop and raise the memory error exception
4. If *(i == 0)* && *pte[1:0] != 2'b11*, stop and raise a MMU translation fault exception
5. If *pte[1:0] == 2'b01*, a block descriptor has been found. If pte.AF == 1, go to step 8
6. If pte.AF == 0, stop and raise a MMU access flag exception
7. Let *i = i + 1.* If *i > 4*, stop and raise a MMU translation fault exception. Let *root = pte.next-level table address[47:12].* Go to step 2
8. A leaf block or page has been found. Check access permissions. Physical address = *pte.output address + page offset*

15.6 MMU Registers

Table 15-6 Special Purpose Registers for TLB Control

Auxiliary Register	Name
0x460	MMU Root Translation Pointer0 Auxiliary Register, MMU_RTP0
0x462	MMU Root Translation Pointer1 Auxiliary Register, MMU_RTP1
0x464	MMU TLB Index Register, MMU_TLB_IDX
0x465	MMU TLB Command Register, MMU_TLB_CMD
0x466	MMU TLB Data 0 Register, MMU_TLB_DATA0
0x467	MMU TLB Data 1 Register, MMU_TLB_DATA1
0x468	MMU Control Register, MMU_CTRL
0x469	MMU Translation Table Base Control Register, MMU_TTBC
0x46A	MMU Attribute Register, MMU_MEM_ATTR
0x6F	MMU Build Configuration Register, MMU_BUILD

These registers may only be accessed in kernel mode. An attempt to access these registers from user mode results in an exception.

15.6.1 MMU Root Translation Pointer0 Auxiliary Register, MMU_RTP0

Address: 0x460

Access: RW

Reset 0x0000_0000

Figure 15-21 MMU_RTP0 Register for MMUv48

Figure 15-22 MMU_RTP0 Register for MMUv52

This auxiliary register defines the physical address of the root of the translation table 0. The root of the table address must be aligned to the size of the first level table.

Before writing to this register, ensure the page table it points to has been updated in memory. Write accesses are serializing.

MMUv48 – 4K:

- **Root table address:** Bits [47: x] of the base address of translation table 0. The value of x is 12, for either value of T0SZ. This address shall be 4KB aligned
- **ASID:** 16-bit address space identifier

MMUv48 – 16K:

- **Root table address:** Bits [47: x] of the base address of translation table 0. The value of x depends on T0SZ:
  - **T0SZ = 16:** First level table is 16B aligned, x = 4
  - **T0SZ = 25:** First level table is 64B aligned, x = 6
- **ASID:** 16-bit address space identifier

MMUv48 – 64K:

- **Root table address:** Bits [47: x] of the base address of translation table 0. The value of x depends on T0SZ:
  - **T0SZ = 16:** First level table is 512B aligned, x = 9
  - **T0SZ = 25:** First level table is 8KB aligned, x = 13
- **ASID:** 16-bit address space identifier

MMUv52 – 64K

- **Root table address:** Bits [51: x] of the base address of translation table 0. The value of x depends on T0SZ:
  - **T0SZ = 12:** First level table is 8KB aligned and therefore x = 13
  - **T0SZ = 22:** First level table is 64KB aligned and therefore x = 16
- **ASID:** 16-bit address space identifier

15.6.2 MMU Root Translation Pointer1 Auxiliary Register, MMU_RTP1

Address: 0x462

Access: RW

Reset 0x0000_0000

Figure 15-23 MMU_RTP1 Register for MMUv48

Figure 15-24 MMU_RTP1 Register for MMUv52

This auxiliary register defines the physical address of the root of the translation table 1. The root of the table address must be aligned to the size of the first level table.

Before writing to this register, ensure the page table it points to has been updated in memory. Write accesses are serializing.

MMUv48 – 4K:

- **Root table address:** Bits [47: x] of the base address of translation table 1. The value of x is 12, for either value of T1SZ. This address shall be 4KB aligned
- **ASID:** 16-bit address space identifier

MMUv48 – 16K:

- **Root table address:** Bits [47: x] of the base address of translation table 1. The value of x depends on T1SZ:
  - **T1SZ = 16:** First level table is 16B aligned, x = 4
  - **T1SZ = 25:** First level table is 64B aligned, x = 6
- **ASID:** 16-bit address space identifier

MMUv48 – 64K:

- **Root table address:** Bits [47: x] of the base address of translation table 1. The value of x depends on T1SZ:
  - **T1SZ = 16:** First level table is 512B aligned, x = 9
  - **T1SZ = 25:** First level table is 8KB aligned, x = 13
- **ASID:** 16-bit address space identifier

MMUv52 – 64K

- **Root table address:** Bits [51: x] of the base address of translation table 1. The value of x depends on T1SZ:
  - **T1SZ = 12:** First level table is 8KB aligned and therefore x = 13
  - **T1SZ = 22:** First level table is 64KB aligned and therefore x = 16
- **ASID:** 16-bit address space identifier

15.6.3 MMU TLB Index Register, MMU_TLB_IDX

Address: 0x464

Access: RW

Reset 0x0000_0000

Figure 15-25 MMU_TLB_IDX 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 `MMU_TLB_IDX` register is mapped as shown in Table 15-7.

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. For more information, see Table 15-8.

Table 15-7 MMU_TLB_IDX Addresses

Access Type	Address
L2 TLB	0x000 to 0x7FF
ITLB	0x1000 to 0x1000 + number of ITLB entries
DTLB	0x1100 to 0x1100 + number of DTLB entries
MMU Translation Cache	■ Level 1 page table translation cache: 0x2000 to 0x2000 + size of L1 ■ Level 2 page table translation cache: 0x2100 to 0x2100 + size of L2 ■ Level 3 page table translation cache: 0x2200 to 0x2200 + size of L3

RC

This field is read-only. This three-bit field is set by the hardware indicating results from certain commands. Writes to this field are ignored.

Table 15-8 Result Code, Read Only

Code	Description
0x0	Normal operation
0x2	Incompatible size and index detected

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.

15.6.4 MMU TLB Command Register, MMU_TLB_CMD

Address: 0x465

Access: W

Reset: 0x0000_0000

Figure 15-26 MMU_TLB_CMD Register

TLB maintenance operations provide a mechanism to ensure that changes to the page tables in memory are reflected (invalidated) on the TLB. TLB maintenance operations are initiated by performing a write on this auxiliary register. Some TLB maintenance operations require that MMU_TLB_DATA auxiliary registers are set-up before writing the command on the MMU_TLB_CMD register. An attempt to write an invalid command will raise an Instruction Error exception with ECR value 0x020001.

TLB maintenance instructions can be performed when the MMU is disabled.

The following commands are supported:

Table 15-9 MMU_TLB_CMD Register Command List

Command	Name	Description
0x1	`TLBInvalidateAll`	Invalidate all TLB entries
0x2	`TLBRead`	Read the TLB entries into TLBData register
0x3	`TLBInvalidateASID`	Invalidate all TLB entries with the ASID specified in MMU_TLB_DATA0
0x4	`TLBInvalidateAddr`	Invalidate all TLB entries with virtual address matching MMU_TLB_DATA0
0x5	`TLBInvalidateRegion`	Invalidate TLB virtual address region from MMU_TLB_DATA0 to MMU_TLB_DATA1

TLBInvalidateAll:

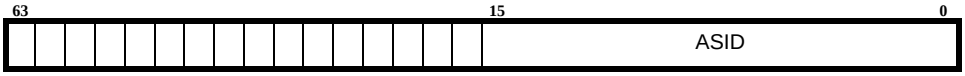
All TLB entries are invalidated, including L1 and L2 TLBs

TLBRead:

This TLB command reads the TLB entry specified in the MMU_TLB_IDX auxiliary register into TLBData auxiliary register. The TLBRead command is primarily using for debug purposes.

TLBInvalidateASID:

The ASID must be programmed in the lower bits MMU_TLB_DATA0 auxiliary register. All non-global TLB entries with an ASID that matches the ASID in MMU_TLB_DATA0 are invalidated. This command affects both L1 and L2 TLBs.



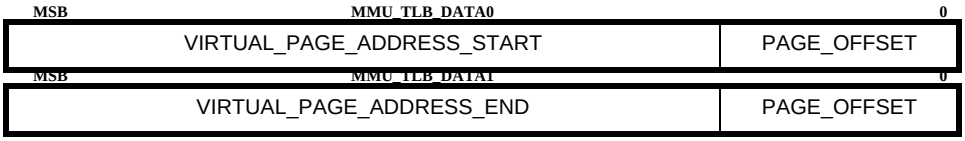
TLBInvalidateAddr:

The virtual address must be programmed in the MMU_TLB_DATA0 auxiliary register. All TLB entries with virtual address matching MMU_TLB_DATA0 are invalidated, regardless of ASID value. This command affects both L1 and L2 TLBs.

TLBInvalidateRegion:

This operation invalidates a virtual address region from MMU_TLB_DATA0 (inclusive) to MMU_TLB_DATA1 (exclusive). This means the MMU page that contains the virtual address programmed in MMU_TLB_DATA1 is not invalidated.

This operation is a no-op when MMU_TLB_DATA0 > MMU_TLB_DATA1.



15.6.5 MMU TLB Data 0 Register, MMU_TLB_DATA0

Address: 0x466

Access: RW

Reset: 0x0000_0000

Figure 15-27 TLB Data Register

The MMU TLB data0 auxiliary register is used by TLB maintenance operations. Some TLB maintenance operations require this auxiliary register is programmed before the TLB command register is written.

15.6.6 MMU TLB Data 1 Register, MMU_TLB_DATA1

Address: 0x467

Access: RW

Reset: 0x0000_0000

Figure 15-28 TLB Data Register

The MMU TLB data1 auxiliary register is used by TLB maintenance operations. Some TLB maintenance operations require this auxiliary register is programmed before the TLB command register is written.

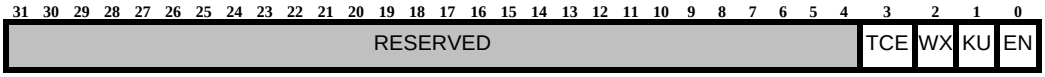
15.6.7 MMU Control Register, MMU_CTRL

Address: 0x468

Access: RW

Reset: 0x0000_0000

Figure 15-29 MMU_CTRL Register



This auxiliary register provides global controls for the MMU. Write to this register are serializing.

Note

- Changing KU or WX bits after the MMU has been enabled, requires an invalidation of the TLBs before writing the new KU and WX values in the MMU_CTRL auxiliary register.
- An SR write to disable the MMU (toggling MMU_CTRL.EN from 1 to 0) must be followed by a SYNC instruction.

Table 15-10 MMU_CTRL Register Field Descriptions

Field	Bit	Description
Enable, EN	[0]	This bit is asserted to enable all MMU functions: address translation and permission checks
Kernel mode access to user mode data, KU	[1]	This bit controls the behavior of load/store instructions in kernel mode accessing a virtual memory address that is also accessible in user-mode. The KU bit affects the access permission (AP) bits as defined: ■ 0x0: Read-write access in kernel mode only ■ 0x1: Read-write access in user mode and read-write access in kernel mode if MMU_CTRL.KU is set ■ 0x2: Read-only access in kernel mode only ■ 0x3: Read-only access in user mode and read-only access in kernel mode if MMU_CTRL.KU is set
Writeable pages execution behavior, WX	[2]	When this bit is de-asserted, pages that are writable at a given privilege level do not have execute permissions at that and higher privilege levels. When this bit is asserted, the execute permissions are not influenced by the AP (access permission) bits
Translation cache enable, TCE	[3]	This bit is asserted to enable the translation cache. This bit is RAZ/IOW in configurations where the translation cache is not present

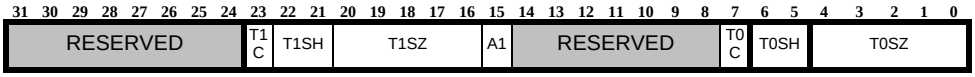
15.6.8 MMU Translation Table Base Control Register, MMU_TTBC

Address: 0x469

Access: RW

Reset: 0x0000_0000

Figure 15-30 MMU_TTBC Register



This auxiliary register provides MMU translation mechanism. Write to this register are serializing.

Note A change to the MMU_TTBC aux register after the MMU has been enabled requires a full TLB invalidation. The entire translation cache is invalidated as a result of the TLB invalidation maintenance instruction.

Table 15-11 MMU _TTBC Register Field Descriptions

Field	Bit	Description
T0SZ	[4:0]	Size of the virtual region mapped by translation table 0. See Table 15-12.
T0SH	[6:5]	Cacheability and shareability attributes for memory associated with translation table 0 walk ■ 0x0: Non-shareable ■ 0x1: Reserved ■ 0x2: Outer-shareable ■ 0x3: Inner-shareable
T0C	[7]	Cacheability attribute for memory associated with translation table 0 walk ■ **0x0:** Uncached ■ 0x1: Cached
A1	[15]	When this bit is de-asserted, pages that are writable at a given privilege level do not have execute permissions at that and higher privilege levels. When this bit is asserted, the execute permissions are not influenced by the AP (access permission) bits
T1SZ	[20:16]	Size of the virtual region mapped by translation table 1. See Table 15-12.

Table 15-11 MMU _TTBC Register Field Descriptions (Continued)

Field	Bit	Description
T1SH	[22:21]	Cacheability and shareability attributes for memory associated with translation table 1 walk ■ 0x0: Non-shareable ■ 0x1: Reserved ■ 0x2: Outer-shareable ■ 0x3: Inner-shareable
T1C	[23]	Cacheability attribute for memory associated with translation table 1 walk ■ **0x0:** Uncached ■ 0x1: Cached

Table 15-12 Legal TS0Z and TS1Z

Field	T0SZ	RTP0 Region	T1SZ	RTP1 Region
MMUv48 – 4K	16	Zero to 2^{48} -1	16	(2^{64} – 2^{48}) to max
	25	Zero to 2^{39} -1	25	(2^{64} – 2^{39}) to max
MMUv52	12	Zero to 2^{52} -1	12	(2^{64} – 2^{52}) to max
	22	Zero to 2^{42} -1	22	(2^{64} – 2^{42}) to max

Note Values other than the ones listed in the preceding table are reserved and are treated as 16 for MMUv48 and 12 for MMUv52 respectively.

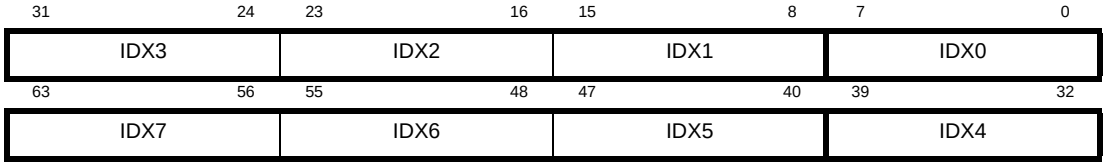
15.6.9 MMU Attribute Register, MMU_MEM_ATTR

Address: 0x46A

Access: RW

Reset 0x0000_0000_0000_0000

Figure 15-31 MMU_MEM_ATTR



Some memory attributes of a page are not directly stored in the page tables. The page table only contain a 3-bit pointer to a field of an auxiliary register.

The descriptor only contains a 3-bit pointer to one of the 8 available fields of the MEM_ATTR auxiliary register.

Each field is 8-bit wide and defines volatile or normal memory regions. The top four bits define the outer cache attributes. These are exported on the system bus (AXI/ACE interface). The bottom four bits are attributes for the volatile or normal memory regions.

The B attribute for this region is conveyed on the AWCACHE[0] bufferable attribute.

Note The AXI bufferable and the early-write-buffer-acknowledge attributes are configured independently. This allows non-posted writes to be buffered in the write buffer

Volatile Memory Region

A volatile memory region is defined by programming bit[0] to 0.

Table 15-13 Volatile Memory Region

MEM_ATTR[7:4]	MEM_ATTR[3:0]	Description
000B	x000	Volatile, no early write acknowledge, strict ordering
000B	x010	Volatile, early write acknowledge, strict ordering
000B	x100	Volatile, no early write acknowledge, relaxed ordering
000B	x110	Volatile, early write acknowledge, relaxed ordering
Others: Reserved, treated as 000B	x: Reserved, treated as 0	

Normal Region

A normal memory region is defined by programming bit[0] to 1. The instruction cacheability is defined by bit[1].

Table 15-14 Normal Memory Attributes

MEM_ATTR[7:4]	MEM_ATTR[3:0]	Description
000B	0x01	Normal memory (data) uncached
000B	0x11	Normal memory (instruction) uncached, (data) uncached
0WR1	1x11	Normal memory (instruction) uncached, inner (data) cache-write back
0WR1	1x01	Normal memory inner (data) cached-write back
Others: Reserved, treated as 000B	x: Reserved, treated as 0	

The W, R and B are outer attributes reflected on the AxCACHE AXI4/ACE system bus. Where, R=Outer Read-allocate policy, W=Outer Write-allocate policy

Table 15-15 AXI4 Attributes

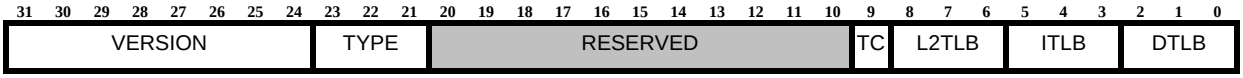
Memory Type	AxCACHE[3] Allocate	AxCACHE[2] Other allocate	AxCACHE[1] Modifiable	AxCACHE[0] Bufferable
Volatile memory	0	0	0	B
Normal memory uncached	0	0	1	B
Normal memory, inner write-back cacheable	W	R	1	1

15.6.10 MMU Build Configuration Register, MMU_BUILD

Address: 0x6F

Access: R

Figure 15-32 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 15-16 MMU_BUILD Register Fields

Field	Bit	Description
DTLB	[2:0]	Indicates the number of DTLB/ µDTLB entries ■ `0x2`: 8 entries ■ `0x3`: 16 entries All other values are reserved.
ITLB	[5:3]	Indicates the number of ITLB/ µITLB entries ■ `0x1`: 4 entries ■ `0x2`: 8 entries ■ `0x3`: 16 entries All other values are reserved.
L2TLB	[8:6]	Indicates the number of entries in the four-way associative L2 TLB ■ `0x0`: 256 ■ `0x1`: 512 ■ `0x2`: 1024 ■ `0x3`: 2048
TC	[9]	■ `0x0`: MMU translation cache is not available ■ `0x1`: MMU translation cache is available

Table 15-16 MMU_BUILD Register Fields

Field	Bit	Description
Type	[23:21]	■ ■ 0x1: MMUv48-4K ■ 0x2: MMUv48-16K ■ 0x3: MMUv48-64K ■ 0x4: MMUv52
VERSION	[31:24]	VERSION ■ `0x10`: MMU version for the ARCv3 processors

15.7 Software Sequence For Updating Page Tables

Page tables may be shared by multiple cores and these cores may be running during page table updates. In order to ensure a coherent view of the page tables by all observers, software must follow the sequence specified in this section when updating the page table in memory or changing the MMU_RTP auxiliary register.

The intention of this sequence is to ensure that at no time both the new and old page table entry is visible to different threads of execution.

The steps are defined as follows**:**

1. Perform a store instruction to replace the old page table entry with an invalid entry
2. Execute a DSYNC instruction to ensure the store instruction at step 1 is completed before any subsequent instruction is executed on this thread
3. Perform a TLB invalidate instruction to remove this page table entry from the local TLB, i.e.: TLB from this core
4. Execute a DSYNC instruction to ensure the TLB invalidate instruction at step 3 is completed before any subsequent instruction is executed on this thread
5. Perform an Inter-Processor-Interrupt (IPI) to other cores
6. Perform similar TLB maintenance invalidation on other cores and acknowledge the IPI
7. Write (store) the new translation table entry
8. Execute a SYNC instruction to ensure the new entry is visible to the subsequent instruction fetch

This sequence shall be followed when a page table is shared among multiple threads of execution (e.g.: multiple cores) and the update of the page table involves one of the following:

- Change of virtual or physical address
- Change of the memory or cacheability attributes
- Change of the size of the mapped block (pages to blocks or vice-versa)

The processor may see erroneous or corrupt data if this software sequence is not followed.

This software sequence is not required when the update of the page table involves:

- Upgrading the permissions of an entry. If a core whose TLB has not been updated accesses the old permissions, this core takes a page fault
- Setting the access or dirty bit of an entry

Only the TLB maintenance invalidation is required in the cases above.

Note The page table should not reside in the address space reserved for closely-coupled-memories.

16 ICCM

16.1 ICCM Auxiliary Registers

This section describes the ARCv3 Instruction Closely Coupled Memories (ICCM0 and ICCM1). ICCMs are designed to hold performance-critical code.

The ARCv3 architecture includes the following auxiliary registers and build configuration registers when the instruction close coupled memory is enabled.

The ICCM base address shall be mapped on the lower 4GB of the address space and shall be aligned on a 1MB boundary.

Table 16-1 ICCM Registers

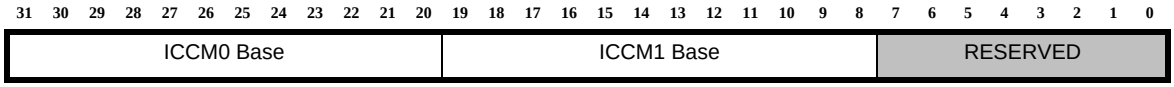
Address	Auxiliary Register Name	Description
0x208	AUX_ICCM	Base address of multiple ICCMs
0x78	ICCM_BUILD	Build configuration register for ICCM

16.1.1 ICCM base address, AUX_ICCM

Address: 0x208

Access: RW

Figure 16-1 AUX_ICCM, base address for ICCM



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 ICCM0 and ICCM1 base addresses are 1 MB aligned. If the ICCM configured size is larger than 1 MB, the base address of the CCM must be aligned with the ICCM size. If ICCM is not configured, this register does not exist.

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

16.1.2 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 16-2 ICCM_BUILD Register

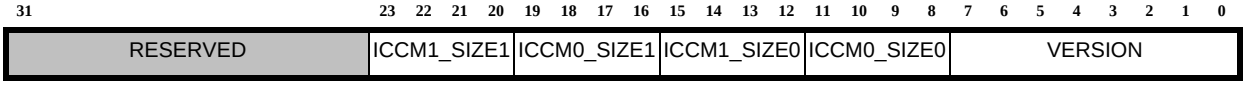


Table 16-2 ICCM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	■ 0x6: ARCv3 implementation
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 16-2 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

16.2 Exceptions

See Table 7-3 on page 204.

17 Instruction Cache

17.1 Instruction Cache Auxiliary Registers

The ARCv3 architecture includes additional auxiliary registers when the instruction cache is configured.

Advanced Instruction-Cache Maintenance Instructions

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 ARCv3 host interface bus or through LR and SR instructions. This capability provides an invaluable debug and test environment when using the instruction cache.

17.2 Auxiliary Registers

Table 17-1 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
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

Table 17-1 Instruction Cache Auxiliary Registers (Continued)

Address	Auxiliary Register Name	Description
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

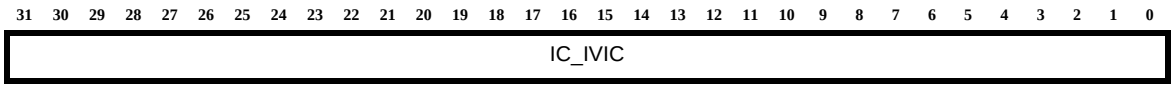
17.2.1 Invalidate Instruction Cache, IC_IVIC

Address: 0x10

Access: W

Reset: 0x0000_0000

Figure 17-1 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.

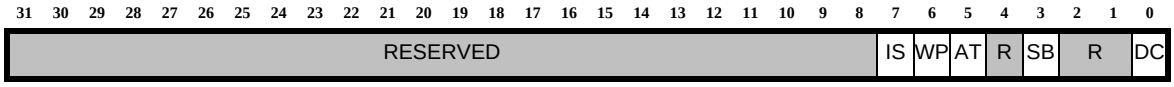
17.2.2 Instruction Cache Control Register, IC_CTRL

Address: 0x11

Access: RW

Reset: 0x0000_0000

Figure 17-2 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 17-2 IC_CTRL Control Flags

Flag Name	Bit	Description	Access Type
DC	[0]	Disable Cache: Enables/Disables the cache ■ `0x0`: Enable Cache ■ `0x1`: Disable Cache	R/W
SB	[3]	Success Bit: Success of last cache operation. The SB bit is updated for some AUX cache maintenance instructions. For example, line lock, line/region invalidate, and debug read.	R
AT	[5]	Address Debug Type: Used for debug purposes for when accessing cache RAMs. ■ `0x0`: Direct Cache RAM Access ■ `0x1`: Cache Controlled RAM Access	R/W
WP	[6]	Instruction Cache way-prediction ■ `0x0`: Disable way-prediction ■ `0x1`: Enable way-prediction	R/W
IS	[7]	Instruction cache maintenance Status: This bit is asserted when an instruction cache maintenance instruction (instruction cache locking or prefetching) is pending. Note: The locking or prefetching of instruction cache lines are non-blocking maintenance instructions. These instructions are completed when the IC_CTRL.IS bit is cleared. It is responsibility of software to check the status of the IS bit after performing a locking or prefetching maintenance instruction.	R

For more information about the IC_CTRL register implementation, see your *ARCv3-based processor Databook*.

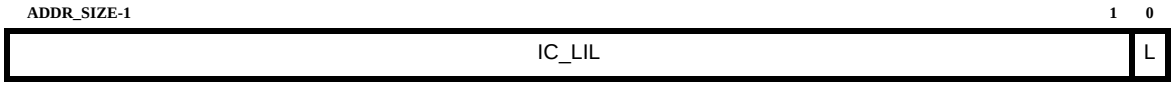
17.2.3 Lock Instruction Cache Line, IC_LIL

Address: 0x13

Access: W

Reset: 0x0000_0000

Figure 17-3 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 $\log_2$(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 $\log_2$(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.

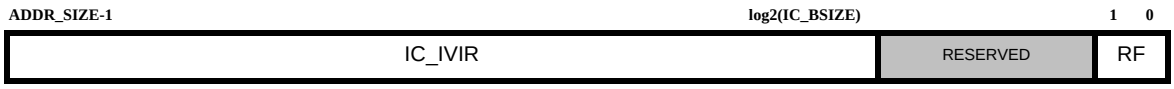
17.2.4 Invalidate Instruction-Cache Start Region, IC_IVIR

Address: 0x16

Access: W

Reset: 0x0000_0000

Figure 17-4 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 $\log_2$(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 `IC_CTRL[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

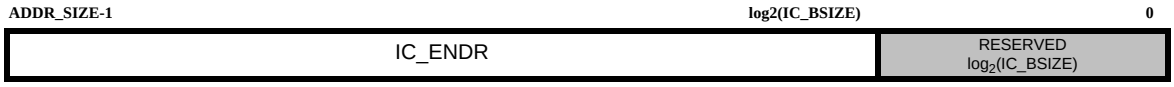
17.2.5 Invalidate Instruction-Cache End Region, IC_ENDR

Address: 0x17

Access: RW

Reset: 0x0000_0000

Figure 17-5 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 for all region operations such as region prefetch, region lock, or invalidation. 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).

17.2.6 Invalidate Instruction-Cache Line, IC_IVIL

Address: 0x19

Access: W

Reset: 0x0000_0000

Figure 17-6 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).

17.2.7 Instruction Cache External-Access Address, IC_RAM_ADDR

Address: 0x1A

Access: RW

Reset: 0x0000_0000

Figure 17-7 IC_RAM_ADDR for Cache Controlled RAM Access (AT==1)

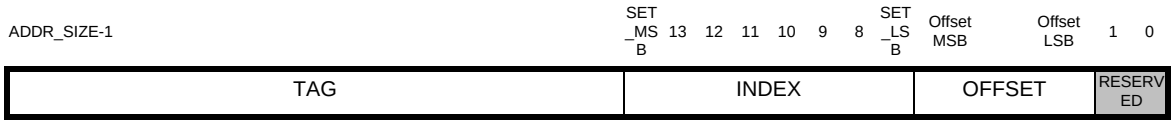


Figure 17-8 IC_RAM_ADDR Register for Direct-Cache-Access (AT==0)

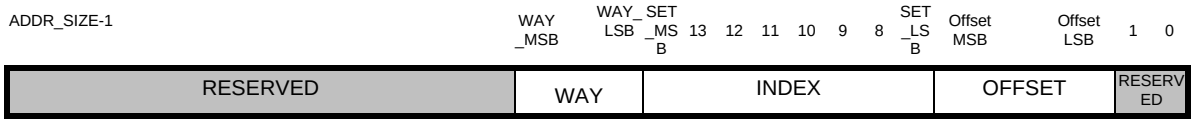
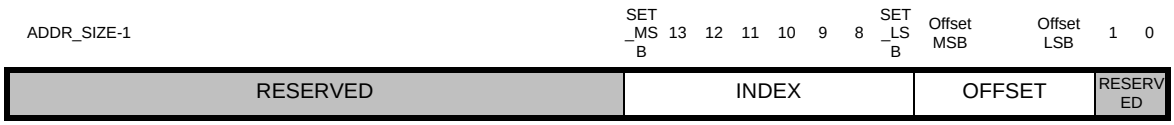


Figure 17-9 IC_RAM_ADDR for Aliased Configurations



The interpretation of the register is defined by the AT (Address Debug Type) flag in the IC_CTRL register. See Table 17-3 for more information.

When the physical address extension is enabled in the MMU, the IC_PTAG_HI register is also updated with the upper bits of the tag.

Table 17-3 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 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_RAM_ADDR 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 17-3 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

17.2.8 Instruction-Cache Tag Access, IC_TAG

Address: 0x1B

Access: RW

Reset: 0x0000_0000

Figure 17-10 IC_TAG Register for Reads

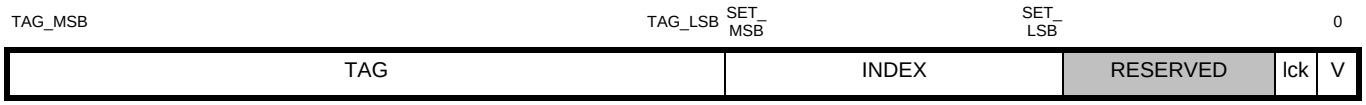


Figure 17-11 IC_TAG Register for Reads in an Aliased Configuration

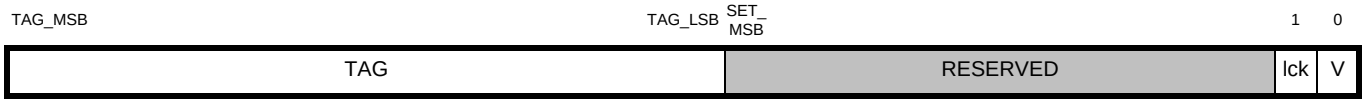
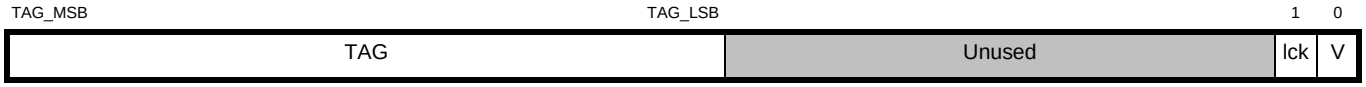


Figure 17-12 IC_TAG Register for Writes



Note

Following are the definitions of the various fields in this register:

- SET_LSB = Log_2(IC_BSIZE)
- SET_MSB = Log_2(IC_SIZE/IC_WAYS) - 1
- TAG_LSB = Log_2(IC_SIZE/IC_WAYS)
- TAG_MSB = PC_SIZE - 1

Reading from or writing to this register allows the programmer to access the valid bit, lock bit, and tag of a cache line specified by the IC_RAM_ADDR register and the AT flag found in IC_CTRL. An SR to the IC_RAM_ADDRESS register also triggers an update of the IC_TAG and IC_DATA registers.

When the physical address extension is enabled in the MMU, the IC_PTAG_HI register is also updated with the upper bits of the tag.

Table 17-4 describes the IC_TAG register behavior.

Table 17-4 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 17-4 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.

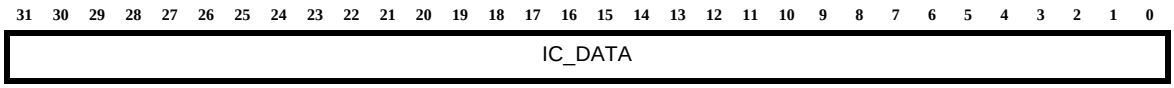
17.2.9 Instruction Cache Data Access, IC_DATA

Address: 0x1D

Access: RW

Reset: 0x0000_0000

Figure 17-13 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 17-5 describes the behavior of the IC_DATA register in a RAM direct access mode and a cache controlled mode.

Table 17-5 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 17-5 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

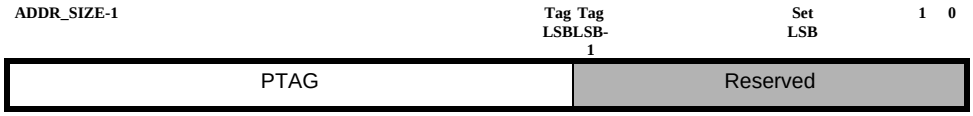
17.2.10 Instruction Cache External Access RAM Address Physical Tag Format, IC_PTAG

Address: 0x1E

Access: RW

Reset: 0x0000_0000

Figure 17-14 IC_PTAG Register



When the ARCv3 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, for MMUv32] 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.

For configurations with MMUv32, the IC_PTAG_HI register exists and provides physical address bits above bit 31. Program the IC_PTAG_HI register irrespective of MMU status. 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, a region cannot span a 4G address boundary.

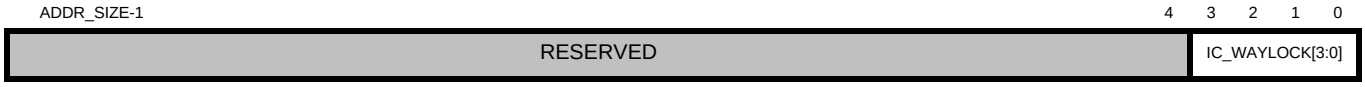
17.2.11 Instruction Cache Way-Lock Register, IC_WAYLOCK

Address: 0x14

Access: RW

Reset: 0x00000000

Figure 17-15 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.

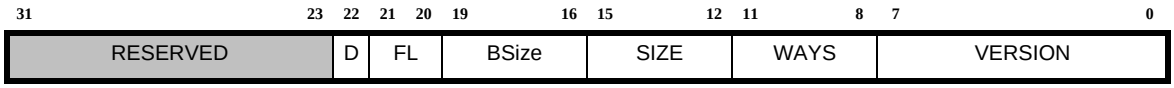
Each bit in the IC_WAYLOCK[3:0] field represents a way. To lock way 0, set the IC_WAYLOCK[0]==1. Similarly, to lock a way, write 1 to the corresponding bit. Each way can be locked independently of the other ways.

17.2.12 Instruction Cache Configuration Register, I_CACHE_BUILD

Address: 0x77

Access: R

Figure 17-16 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 17-6 I_CACHE_BUILD Field Descriptions

Field Name	Bit	Description
VERSION	[7:0]	Version number ■ 0x6: current version, 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 *All other values are Reserved*
WAYS	[11:8]	Cache Associativity ■ 0x1: Two-way set associative ■ 0x2: Four-way set associative *All other values are Reserved*
SIZE	[15:12]	Cache capacity ■ 0x2: 2 Kbytes ■ 0x3: 4 Kbytes ■ 0x4: 8 Kbytes ■ 0x5: 16 Kbytes ■ 0x6: 32 Kbytes ■ 0x7: 64 Kbytes ■ 0x8: 128 Kbytes *All other values are Reserved*
BSize	[19:16]	Block Size, indicates the cache block size in bytes. ■ 0x3: 64 bytes

Table 17-6 I_CACHE_BUILD Field Descriptions (Continued)

Field Name	Bit	Description
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

17.3 Exceptions

See Table 7-3 on page 204.

18 DCCM

18.1 DCCM Auxiliary Registers

This section describes the ARCv3 Data Closely Coupled Memories (DCCM). DCCM is designed to hold performance-critical data.

The ARCv3 architecture includes the following auxiliary registers and build configuration registers when the DCCM is enabled.

Table 18-1 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

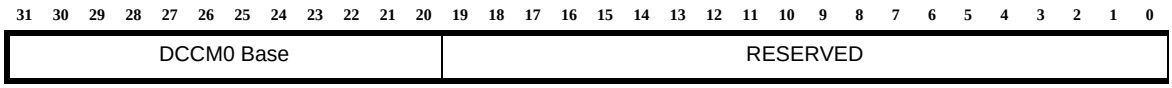
18.1.1 DCCM Base Address, AUX_DCCM

Address: 0x18

Access: RW

Default 0x8000_0000

Figure 18-1 AUX_DCCM, base address for DCCM



When a DCCM is configured in a processor, the AUX_DCCM register identifies the base address of the DCCM. The DCCM base must be aligned to 1 MB. If the DCCM configured size is larger than 1 MB, the base address of the CCM must be aligned with the DCCM size. If DCCM is not configured, this register does not exist.If DCCM is not configured, this register does not exist.

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 or untranslated.
- 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.

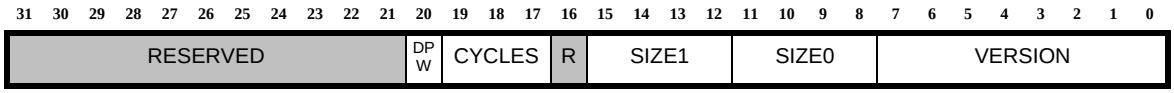
Repositioning the DCCM does not affect the contents of external memory. Similarly, external DMA accesses to addresses that overlap external memory and DCCM do not affect the contents of DCCM.

18.1.2 DCCM Configuration Register, DCCM_BUILD

Address: 0x74

Access: R

Figure 18-2 DCCM_BUILD Register



The DCCM RAM Configuration register (DCCM_BUILD) describes the size and version number of the Data Closely Coupled Memory.

Table 18-2 DCCM_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	■ 0x6: ARCv3 implementation
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

Table 18-2 DCCM_BUILD Field Descriptions (Continued)

Field	Bit	Description
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 `-dccm_mem_banks` 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 `DCCM_BUILD.VERSION==0x4`.
DPW	[20]	This field is always set to 0x1.

18.1.3 Exceptions

See Table 7-3 on page 204.

19 Data Cache

The ARCv3 architecture includes additional auxiliary registers when the `data cache` component is enabled.

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.

19.1 Data Cache Auxiliary Registers

Table 19-1 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	0x0000_30c0
0x4B	Flush Data Cache, DC_FLSH	Flush data cache	-
0x49	Lock Data Cache Line, DC_LDL	Lock data line	-
0x4A	Invalidate Data Line, DC_IVDL	Invalidate data line	-
0x4C	Flush Data Line, DC_FLDL	Flush data line	-
0x4D	Data-Cache Region Start Address, DC_STARTR	Data cache region operation start address	-
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

Table 19-1 Data Cache Auxiliary Registers

0x59	Data Cache Tag Access, DC_TAG	Data cache tag access	0x0000_0030
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.

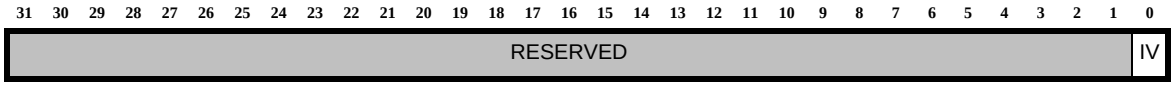
19.1.1 Invalidate Data Cache, DC_IVDC

Address: 0x47

Access: W

Reset: 0x0000_0000

Figure 19-1 DC_IVDC Register



The flush (DC_FLSH) or invalidation (DC_IVDC) of the entire data-cache applies only to the L1 data cache of the core initiating the cache maintenance operation. They are useful operations when preparing a core for a deep sleep/power down state or by the operation system at boot time. This operation ensures that any modified data in the L1 data cache is flushed out before the L1 cache is powered down. Non-coherent cache maintenance operations cannot observe pending operations to same-address locations initiated on other cores, and therefore these non-coherent cache maintenance operations must only be executed at times when software can guarantee that there are no concurrent transactions to any of the addresses that are being flushed. A SYNC instruction is required before and after the SR instruction initiating these cache management operations.

All line and region based data-cache management operations are coherent.

Table 19-2 DC_IVDC Field Descriptions

Flag Name	Bit	Description	Access Type
IV	[0]	Invalidate Data Cache: Invalidates entire data cache ■ `0x0`: No Action ■ `0x1`: Invalidate Data Cache	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.

19.1.2 Data Cache Control Register, DC_CTRL

Address: 0x48

Access: RW

Reset: 0x0000_00C0

Figure 19-2 DC_CTRL Register

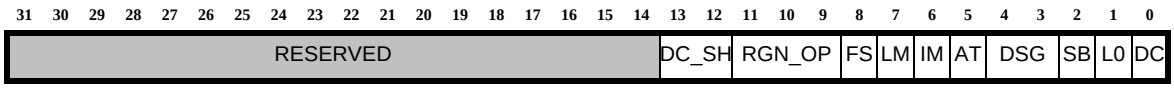


Table 19-3 DC_CTRL Control Flags

Flag Name	Bit	Description	Access Type
DC	[0]	Disable Cache: Enables/Disables the cache 0 – Enable Cache 1 – Disable Cache	R/W
L0	[1]	Disable L0 data cache 0 – Enable L0 Cache 1 – Disable L0 Cache Note: This field is reserved in configuration without a L0 data cache.	R/W
SB	[2]	Success Bit: Success of last cache operation SB is for compatibility with L1 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

Table 19-3 DC_CTRL Control Flags

DSG	[4:3]	DSG (Disabled Store Graduation): These bits are used to disable the store graduation feature. It is possible to disable the graduation of cached and uncached stores independently `0x0`: Store graduation is enabled `0x1`: Disable store graduation of uncached stores `0x2`: Disable store graduation of cached stores `0x3`: Disable graduation of all stores Note: In configurations without D-Cache, store graduation is always enabled.	R/W
AT	[5]	Address Debug Type: Used for debug purposes for when accessing cache RAMs. `0` – Direct Cache RAM Access `1` – Cache Controlled RAM Access	R/W
IM	[6]	Invalidate Mode: Selects the invalidate type `0` – Invalidate data cache only `1` – Invalidate data cache and flush dirty entries; this operation is also referred to as a purge operation.	R/W
LM	[7]	Lock Mode: Selects the effect of a flush command on a locked entry `0` – Disable flush on locked entry `1` – Enable flush on locked entry	R/W
FS	[8]	Flush Status: Status of the data-cache flush mechanism `0` – Idle `1` – Flush operation in progress	R
RGN_OP	[11:9]	Region Operation. Selects the operation to be performed on the address region specified by the DC_STARTR and DC_ENDR registers.This register and the DC_ENDR register must be set up first before writing DC_STARTR with the region start address which triggers the operation. `0x0`: Region flush `0x1`: Region invalidate `0x2:0x7`: Reserved	R/W
DC_SH	[13:12]	DC_SH (Shareability attribute for d-cache maintenance instructions): This field specifies the shareability attribute of the physical address (or physical address region) of coherent Data Cache maintenance instructions `0x0`: Non-shareable (private) `0x1`: Reserved (treated as 0x3) `0x2`: Outer-shareable `0x3`: Inner-shareable	R/W

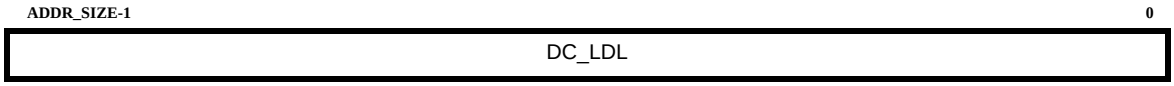
19.1.3 Lock Data Cache Line, DC_LDL

Address: 0x49

Access: W

Reset: 0x0000_0000

Figure 19-3 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 19-3.

For more information, see your ARCv3-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.

19.1.4 Invalidate Data Line, DC_IVDL

Address: 0x4A

Access: W

Reset: 0x0000_0000

Figure 19-4 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.

19.1.5 Flush Data Cache, DC_FLSH

Address: 0x4B

Access: W

Reset: 0x0000_0000

Figure 19-5 DC_FLSH Register

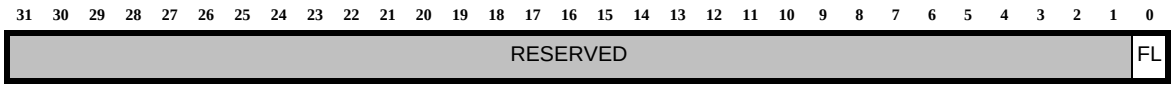


Table 19-4 DC_FLSH Control Flags

Flag Name	Bit	Description	Access Type
FL	[0]	Flush Data Cache:- Flush entire data cache ■ `0x0`: No Action ■ `0x1`: Flush Data Cache	W

The DC_FLSH function is used to update main memory with the most recent copy of the data-cache contents. This is very useful for shared-memory architectures, where other processors require updated memory contents.

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.

19.1.6 Flush Data Line, DC_FLDL

Address: 0x4C

Access: W

Reset: 0x0000_0000

Figure 19-6 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.

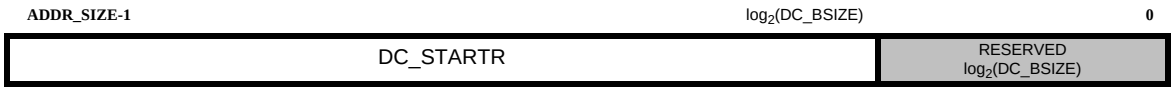
19.1.7 Data-Cache Region Start Address, DC_STARTR

Address: 0x4D

Access: W

Reset: 0x0000_0000

Figure 19-7 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.

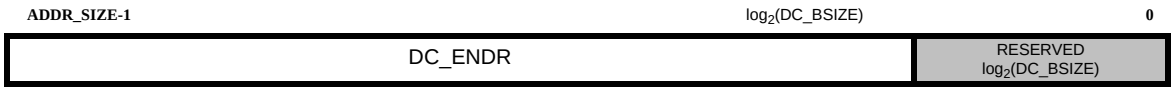
19.1.8 Data-Cache Region End Address, DC_ENDR

Address: 0x4E

Access: RW

Reset: 0x0000_0000

Figure 19-8 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 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.

19.1.9 Data Cache External Access Address, DC_RAM_ADDR

Address: 0x58

Access: RW

Reset: 0x0000_0000

Figure 19-9 DC_RAM_ADDR for Cache Controlled RAM Access (AT==1)

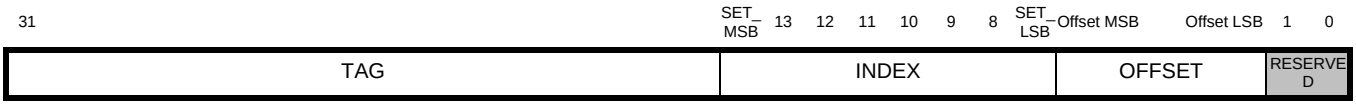
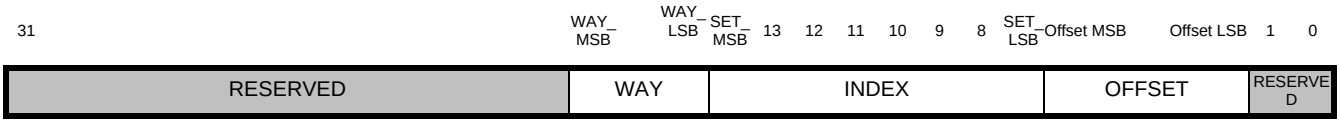


Figure 19-10 DC_RAM_ADDR Register for Direct-Cache-Access (AT==0)



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 979. 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 19-5 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_RAM_ADDR 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 19-5 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

19.1.10 Data Cache Tag Access, DC_TAG

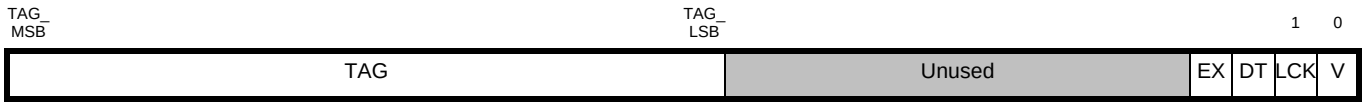
Address: 0x59

Access: RW

Reset: 0x0000_0030; Configuration-dependent

Figure 19-11 DC_TAG for Reads

Figure 19-12 DC_TAG for Writes



Note

Following are the definitions of the various fields in this register:

- TAG_LSB = $\log_2$(DC_SIZE/DC_WAYS)
- TAG_MSB = ADDR_SIZE - 1

Reading from or writing to this register allows the programmer to access the valid bit, lock bit, dirty bit, and tag of a cache line address specified in the DC_RAM_ADDR register. In direct mode, an SR to this register updates the register and has the side-effect of writing the new value to the tag RAM at the location specified by the DC_RAM_ADDR register. A write transaction to the DC_RAM_ADDRESS also triggers an update of the DC_TAG and DC_DATA registers.

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 19-6 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 19-6 DC_TAG Control Flags and Update Conditions (Continued)

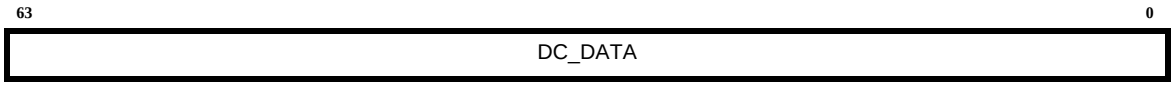
Flag Name	Bit	Description	Flag Update Conditions	Access Type
LCK	[1]	Lock Flag: Lock flag associated with the cache line ■ 0 – Cache line not locked ■ 1 – Cache line locked	Programmer updates to DC_TAG DC_LDL command DC_IVDL command DC_IVDC command	R/W
DT	[2]	Dirty Flag: Dirty flag associated with the 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 Note: This field is only available in configuration with the coherency unit. Otherwise, this field is RAZ, IOW. When present, the reset value of this bit is 0.	Programmer updates to DC_TAG DC_IVDL DC_IVDC Cache line replacement	R/W
SH	[5:4]	SH: Shareability attribute, as defined by MMU/MPU ■ 0x0: Non-shareable (private) ■ 0x1: Reserved (treated as 0x3) ■ 0x2: Outer-shareable ■ 0x3: Inner-shareable Note: This field is only available in configurations with the coherency unit or shared (L2) cache. Otherwise, this field is RAZ, IOW. When present, the reset value of this bit is 0x3.		R

19.1.11 Data Cache Data Access, DC_DATA

Address: 0x5b

Access: RW

Figure 19-13 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.

19.2 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 ARCv3-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 ARCv3 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 ARCv3-based system.

Table 19-7 Build Configuration Registers

Number	Name	Access	Description
0x72	Data Cache Configuration Register, D_CACHE_BUILD	R	Build configuration for: data cache

19.2.1 Data Cache Configuration Register, D_CACHE_BUILD

Address: 0x72

Access: R

Figure 19-14 D_CACHE_BUILD

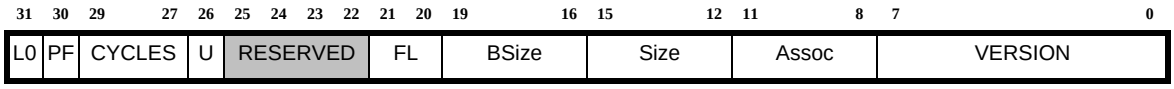


Table 19-8 D_CACHE_BUILD Field Descriptions

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: ARCv3 with fixed number of cycles ■ 0x5: ARCv3 with configurable cycles *All other values are reserved*
Assoc	[11:8]	Cache Associativity. All non-enumerated values are reserved. ■ 0x1: Two-way set associative
Size	[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 ■ 0x8: 128 Kbytes
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

Table 19-8 D_CACHE_BUILD Field Descriptions

Field Name	Bit	Description
FL	[21:20]	Feature Level, indicates locking and debug feature level ■ `0x2`: Lock, flush, invalidation, and advanced debug features are supported ■ `0x3`: *reserved*
U	26	Indicates whether the data cache contains uncached regions. ■ `0x0`: Data cache does not include uncached regions.
CYCLES	[29:27]	■ `0x0`: Not present ■ `0x1`: single cycle memory ■ `0x2`: two-cycle memory; always includes four banks All other values are reserved.
PF	[30]	Indicates whether the prefetch hardware exists in the processor. ■ `0x0`: No prefetch hardware is configured ■ `0x1`: Prefetch hardware is configured.
L0	[31]	Indicates whether the critical pointer cache exists in the processor. ■ `0x0`: No critical pointer cache is configured ■ `0x1`: Critical pointer cache is configured.

19.2.2 Exceptions

See Table 7-3 on page 204.

19.3 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. A region maintenance operation depends on the size of the address region. When the size of the address region is larger than the size of the cache, the hardware searches all sets and ways of the data-cache. This mechanism effectively establishes an

upper time limit for region based operations, regardless of the size of the region. When the size of the address region is smaller than the size of the cache, the hardware searches all addresses in the region, from DC_STARTR to DC_ENDR.

19.3.1 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.

19.3.2 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.

19.3.3 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.

20 Error Protection

The ARCv3 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

20.1 Auxiliary Registers

Table 20-1 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

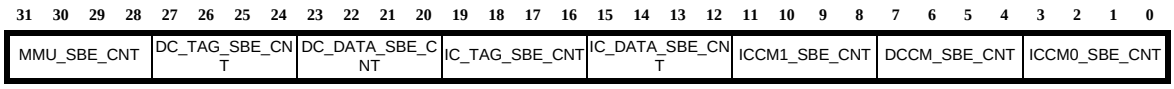
20.1.1 ECC SBE Counter Register, ECC_SBE_COUNT

Address: 0x3C

Access: RW

Reset 0x0000_0000

Figure 20-1 ECC_SBE_COUNT Register



This register records the number of single-bit errors that have been corrected in the memories. Certain bit fields exist only when the related component is present in the build.

Table 20-2 ECC_SBE_COUNT Bit Field Description

Field	Bit	Description
ICCM0_SBE_CNT	[3:0]	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.
DCCM_SBE_CNT	[7:4]	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.

Table 20-2 ECC_SBE_COUNT Bit Field Description

ICCM1_SBE_CNT	[11:8]	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]	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.
IC_TAG_SBE_CNT	[19:16]	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.

Table 20-2 ECC_SBE_COUNT Bit Field Description

DC_DATA_SBE_CNT	[23:20]	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]	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]	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.

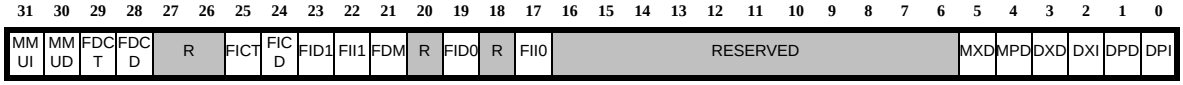
20.1.2 Error Protection Hardware Control Register, ERP_CTRL

Address: 0x3F

Access: RW

Reset 0x00000000

Figure 20-2 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 20-3 ERP_CTRL Bit Field Description

Field	Bit	Description
DISABLE_PROT_IFU (DPI)	[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]	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.
DISABLE_EXCP_IFU (DXI)	[2]	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.

Table 20-3 ERP_CTRL Bit Field Description

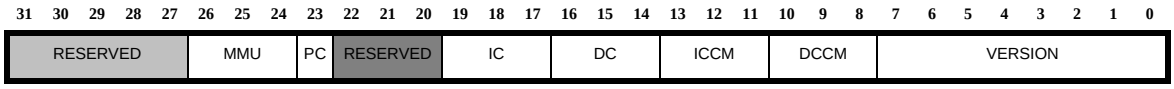
Field	Bit	Description
DISABLE_EXCP_DMP (DXD)	[3]	Control bit for enabling/disabling protection exceptions on DMP; A value of 0 enables exceptions, and 1 disables exceptions for two-bit data and address errors. On reset, the control bits are set to 0; that is exceptions are enabled.
ECC_MMU_DISABLE (MPD)	[4]	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]	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]	Set to 1 indicates double-bit error on ICCM0 instruction word; Write 0 to clear this bit.
FLT_ECC_ICCM0_DATA (FID0)	[19]	Set to 1 indicates error on ICCM0 DMP data word; write 0 to clear this bit.
FLT_ECC_DCCM_DATA (FDM)	[21]	Set to 1 indicates double-bit error on DCCM data word; write 0 to clear this bit.
FLT_ECC_ICCM1_INST_DATA (FII1)	[22]	Set to 1 indicates double-bit error on ICCM1 instruction word; write 0 to clear this bit.
FLT_ECC_ICCM1_ DATA (FID1)	[23]	Set to 1 indicates error on ICCM1 DMP data word; write 0 to clear this bit.
FLT_ECC_ICACHE_INST_DATA (FICD)	[24]	Set to 1 indicates: double-bit error on ICACHE instruction word; write 0 to clear this bit.
FLT_ECC_ICACHE_TAG_DATA (FICT)	[25]	Set to 1 indicates double-bit error on ICACHE_TAG; write 0 to clear this bit.
FLT_ECC_DCACHE_INST_DATA (FDCD)	[28]	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]	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]	Set to 1 indicates double-bit error accessing NTLB during DTLB update; write 0 to clear this bit.
FLT_ECC_MMU_NTLB_ITLB (MMUI)	[31]	Set to 1 indicates double-bit error accessing NTLB during ITLB update; write 0 to clear this bit.

20.1.3 Error Protection Configuration Register, ERP_BUILD

Address: 0xC7

Access: R

Figure 20-3 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 20-4 lists the field descriptions:

Table 20-4 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. ■ `ecc_syndrome_option` 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.

Table 20-4 ERP_BUILD Register

Field	Bit	Description
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. Note: ICCM0 and ICCM1 must have the same error protection configuration.
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.

Table 20-4 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. ■ `0`: indicates EDC protection is not configured ■ `1`: indicates EDC protection is configured When EDC protection is configured, single and double-bit errors are detected and an external signal `edc_pipe_err` is asserted high. Exceptions are not generated for these errors. This bit reflects the value of the configuration option `-pipe_edc_option`.
MMU	[26:24]	Indicates ECC on MMU. ■ `000`: indicates no protection is enabled ■ `001`: indicates 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.

20.2 Exceptions

See Table 7-3 on page 204.

21 Peripheral Bus

The peripheral region is an optional component that allows simple memory-mapped peripherals to be connected to the processor core, using a dedicated bus. The private peripheral interface runs at the core frequency. The size of the private peripheral aperture is configurable at build-time and programmable at run time. The attributes of the loads and store instructions mapped onto these interfaces are programmed on the MMU or MPU. In configurations with virtual memory, the peripheral apertures are mapped onto the physical address space.

Note that a (configurable) shared peripheral interface is also supported by the cluster network.

21.1 DMP Auxiliary Registers

Table 21-1 DMP Registers

Address	Auxiliary Register Name	Description
0x26A	PER_BASE	Peripheral 0 base address
0x26B	PER_SIZE	Peripheral 0 size
0xC3	DMP_PER_BUILD	Peripheral 0 region build configuration register

21.1.1 Private Peripheral Aperture Base Address, PER_BASE

Address: 0x26A

Access: RW

Default 0xF000_0000

Figure 21-1 PER_BASE

This register defines the base address of the private peripheral aperture. The base address is aligned on a 1MB boundary.

Table 21-2 DMP_PER_BUILD Field Descriptions

Field	Bit	Description
Base	[31:20]	Defines the base address of the peripheral 0 aperture. The base address is aligned on a 1MB boundary.

21.1.2 Private Peripheral Aperture Size, PER_SIZE

Address: 0x26B

Access: RW

Default Configuration dependent

Figure 21-2 PER_SIZE

This register defines the size peripheral 0 aperture in MB. The size of the aperture is specified in integer multiples of 1Mbytes.

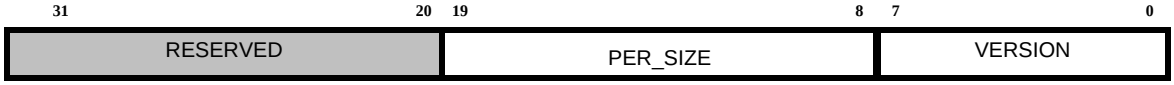
Note The peripheral aperture range is defined as: PER_BASE =< PER < PER_BASE + PER_SIZE.

21.1.3 Peripheral Region Build Configuration Register, DMP_PER_BUILD

Address: 0xC3

Access: R

Figure 21-3 DMP_PER_BUILD Register



This register indicates the presence of a peripheral region, and its build-time configuration.

Table 21-3 DMP_PER_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Version of the peripheral component. All non-enumerated values are reserved. `0x3`: current version
PER_SIZE	[19:8]	Size of the private peripheral aperture (in MB)

21.2 Programming Notes

- The peripheral interface do not support cached accesses. It is a programming error to map a cached region onto the peripheral interface. The processor takes a memory error exception if a cached LD/ST instruction is attempted on a peripheral interface.
- Avoid programming a peripheral aperture that straddles memory boundaries.
- The peripheral apertures are only visible to LD/ST instructions. The Instruction Fetch Unit (IFU) does not access these apertures.
- A best practice is to program the peripheral apertures as non-fetch/non-execute in the Memory Protection Unit or Memory Management Unit
- The attributes of a memory region can be programmed in the MMU or MPU.

22 Write Buffer

22.1 Introduction

The write buffer is an optional component that buffers uncached stores and provides an early write response to the core.

22.2 Configuring the Write Buffer

For information about configuring the write buffer, see the *ARC HS Series Databook*.

22.3 Write Buffer Auxiliary Registers

Table 22-1 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

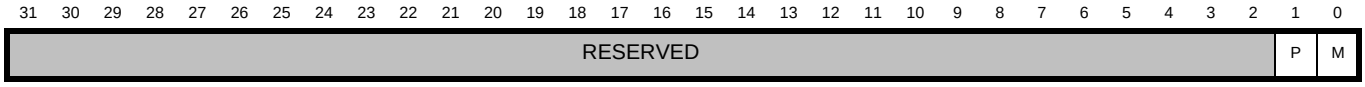
22.3.1 Write Buffer Control Register, WB_CTRL

Address: 0x20C

Access: RW

Reset {strict_ordering, disable} = {1, 0}. Configuration-dependent

Figure 22-1 WB_CTRL



This register allows you to control the enable/disable the write buffers on the peripheral0 and memory. After reset, the write buffer is disabled.

Note A **DSYNC** instruction is required before enabling or disabling the Write Buffer.

Table 22-2 WB_CTRL Register Field Description

Field	Bit	Description
M	[0]	Disable write buffer for memory ■ `0x0`: enables the memory write buffer ■ `0x1`: disables the memory write buffer
P	[1]	Disable write buffer for private peripheral ■ `0x0`: enables the peripheral 0 write buffer ■ `0x1`: disables the peripheral 0 write buffer

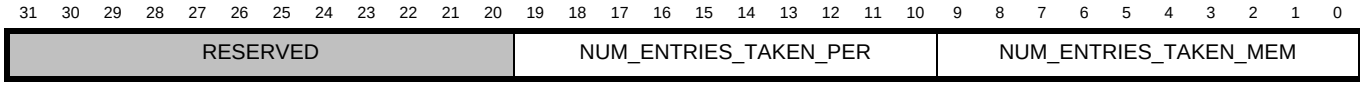
22.3.2 Write Buffer Status Register, WB_STATUS

Address: 0x20D

Access: R

Reset 0x0

Figure 22-2 WB_STATUS



This register records the number of write transactions that are pending in the write buffer.

Table 22-3 WB_STATUS Register Field Description

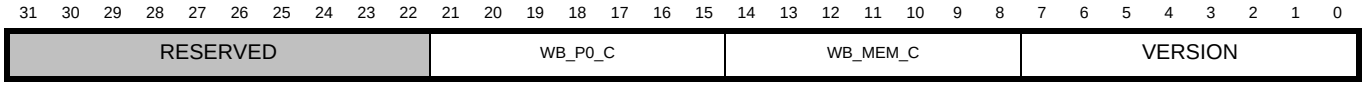
Field	Bit	Description
NUM_ENTRIES_TAKEN_MEM	[9:0]	Number of memory write transactions pending in the write buffer.
NUM_ENTRIES_TAKEN_PER0	[19:10]	Number of private peripheral write transactions pending in the write buffer.

22.3.3 Write Buffer BCR Register, WB_BUILD

Address: 0xFD

Access: R

Figure 22-3 WB_BUILD



This register specifies the write buffer configuration.

Table 22-4 WB_BUILD Register Field Description

Field	Bit	Description
VERSION	[7:0]	■ 0x2: ARCv3 version
WB_MEM_C	[14:8]	Write buffer memory capacity ■ `0x0`: no write buffer present ■ `0x1`: 16 entries ■ `0x2`: 32 entries ■ `0x3`: 64 entries ■ All other values are reserved
WB_P0_C	[21:15]	Write buffer for private peripheral capacity ■ `0x0`: no write buffer present ■ `0x1`: 16 entries ■ `0x2`: 32 entries ■ `0x3`: 64 entries ■ All other values are reserved

23 Branch Prediction Unit

23.1 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.

23.2 Branch Predication Unit Register Interface

Table 23-1 BPU Registers

Address	Auxiliary Register Name	Description
0x480	BPU_FLUSH	Flush and initialize branch predictor register
0x481	BPU_CTRL	BPU control register
0x482	BPU_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_DATA	Prediction Table RAM Data (Predictions)
0xC0	BPU_BUILD	BPU build configuration register

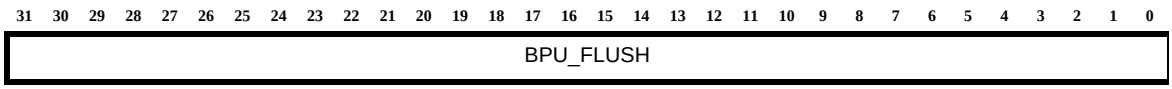
23.2.1 Flush and Initialize Branch Predictor Register, BPU_FLUSH

Address: 0x480

Access: W

Reset: 0x0

Figure 23-1 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.

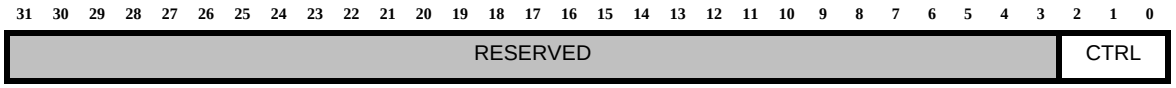
23.2.2 BPU Control Register, BPU_CTRL

Address: 0x481

Access: RW

Reset 0x0

Figure 23-2 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 23-2 BPU_CTRL Field Description

Field	Bits	Description
CTRL[2:0]	[2:0]	■ 3'b000: Normal operation ■ 3'b001: Predict taken. Freeze GHR and do not update prediction tables. Refill Branch Cache ■ 3'b010: Predict not taken. Freeze GHR and do not update prediction tables. Refill Branch Cache ■ 3'b011: Predict taken with freeze. Freeze GHR and do not update prediction tables. Don't refill Branch Cache ■ 3'b100: Predict not taken with freeze. Freeze GHR and do not update prediction tables. Don't refill Branch Cache ■ 3'110: Freeze and don't predict. Same as predict not taken with freeze. ■ 3'b111: Predict with freeze. Do not refill Branch Cache ■ 3'b101: Reserved

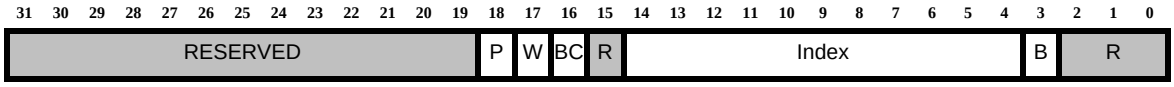
23.2.3 Branch Cache RAM Address Register, BPU_RAM_ADDR

Address: 0x482

Access: RW

Reset 0x0000_0000

Figure 23-3 BPU_RAM_ADDR Register



Use this register to access the branch cache. The effects of reading and writing to this register are summarized in Table 23-3.

Table 23-3 BPU_RAM_ADDR Field Description

Field	Bit	Description
B	[3]	Specifies an even or odd set index
Index	[14:4]	Specifies the set index of the BPU tag/data memory, or the set index of the Prediction memory. The value of IDX_MSB is log2(BR_PT_ENTRIES) and therefore it is configuration dependent. Examples: ■ BR_PT_ENTRIES = 2048 (minimum value) -> IDX_MSB = 11. Bits [14:12] are Reserved (IOW/RAZ) ■ BR_PT_ENTRIES = 16384 (maximum value) -> IDX_MSB = 14
BC	[16]	Memory selection ■ `0x0`: Read PT memory ■ `0x1`: Read BC memory
W	[17]	Specifies the way of the branch cache (only applicable for BC reads) ■ `0x0`: Read way 0 ■ `0x1`: Read way 1
P	[18]	Specifies one of the two predictions per fetch block (only applicable for BC reads) ■ `0x0`: Prediction 0 ■ `0x1`: Prediction 1

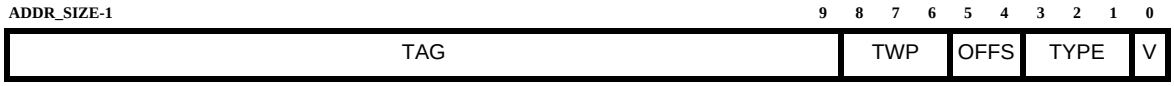
23.2.4 Branch Cache Information Register, BC_RAM_INFO

Address: 0x483

Access: RW

Reset 0x0000_0000

Figure 23-4 BC_RAM_INFO Register



This register provides access to branch cache information.

Table 23-4 BC_RAM_INFO Field Description

Field	Bit	Description
V	[0]	Tag valid bit ■ `0x0`: not valid ■ `0x1`: valid
TYPE	[3:1]	Branch prediction type ■ `0x0`: No prediction ■ `0x1`: Zero-Overhead-Loop prediction ■ `0x2`: Conditional ■ `0x3`: Unconditional ■ `0x4`: Conditional call ■ `0x5`: Unconditional call ■ `0x6`: Conditional return ■ `0x7`: Unconditional return
OFFS	[5:4]	Prediction offset in the fetch block
TWP	[8:6]	I-Cache way prediction ■ 2-way I-Cache: twp[7:6], where twp[7] indicates a valid way prediction and twp[6] indicates I-cache way ■ 4-way I-Cache: twp [8:6], where twp[8] indicates a valid way prediction and twp[7:6] indicates I-cache way
TAG	[ADDR_SIZE-1:9]	Branch address tag

Table 23-5 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.
0x2	BR_CONDITIONAL	Conditional and predicted branch	BBIT0, BBIT1, Bcc, BRcc, Jcc, Bcc_S, BREQ_S, BRNE_S
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		

23.2.5 Branch Cache Branch Target Address Register, BC_RAM_BTA

Address: 0x484

Access: RW

Reset 0x0

Figure 23-5 BC_RAM_BTA Register

This register provides access to branch cache RAM that stores the branch target address.

Table 23-6 BC_RAM_BTA Field Description

Field	Bit	Description
BTA	[ADDR_SIZE-1:1]	Branch cache branch target address

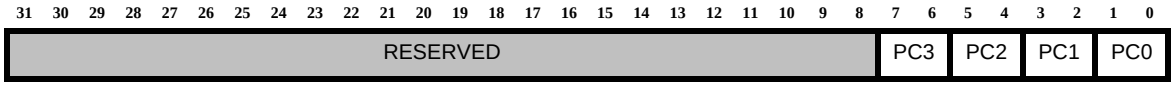
23.2.6 Prediction Table DATA Register, PT_RAM_DATA

Address: 0x485

Access: RW

Reset 0x0

Figure 23-6 PT_RAM_DATA Register



This register provides access to prediction table data.

Table 23-7 PT_RAM_DATA Bit Field Description

Field	Bit	Description
PC0	[1:0]	{Prediction, Confidence} pair for prediction at offset 0
PC1	[3:2]	{Prediction, Confidence} pair for prediction at offset 1
PC2	[5:4]	{Prediction, Confidence} pair for prediction at offset 2
PC3	[7:6]	{Prediction, Confidence} pair for prediction at offset 3

23.2.7 Branch Prediction Unit Configuration Register, BPU_BUILD

Address: 0xC0

Access: R

Figure 23-7 BPU_BUILD Register

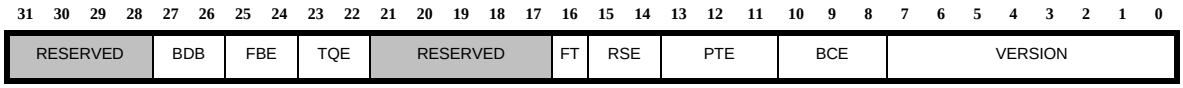


Table 23-8 lists the field descriptions.

Table 23-8 BPU_BUILD Field Descriptions

Field	Bits	Description
VERSION	[7:0]	Branch Prediction Unit version ■ 0x08: First version of branch prediction unit in ARCv3-based processors.
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.
FT	[16]	Use full tag in branch cache ■ 0x1: Use full size tag in branch cache.

Table 23-8 BPU_BUILD Field Descriptions (Continued)

Field	Bits	Description
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 ■ 0x3: 4 All other values reserved.
BDB	[27:26]	Indicates support for BPU debug capability: auxiliary access to BPU memories ■ 0x1: Support for debugging BPU All other values reserved.

23.3 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.

23.3.1 Best Practices to Follow When Debugging BPU

- The best way to use this feature is to rely on the debugger to read the contents of the BPU memories
- The understanding of the G-share prediction is required to understand the content of the BPU prediction table

23.3.2 How to Dump Branch Cache and Prediction Table Data

- A write to the BPU_RAM_ADDR register initiates a read from the Branch Cache (BPU_RAM_ADDR.BC = 1) or from the Branch Prediction Table (BPU_RAM_ADDR.BC = 0).
- The content read from the Branch Cache is written into the BC_RAM_INFO and BC_RAM_BTA registers.
- The content read from the Branch Prediction Table is written into the PT_RAM_DATA registers

23.3.3 Usage Examples

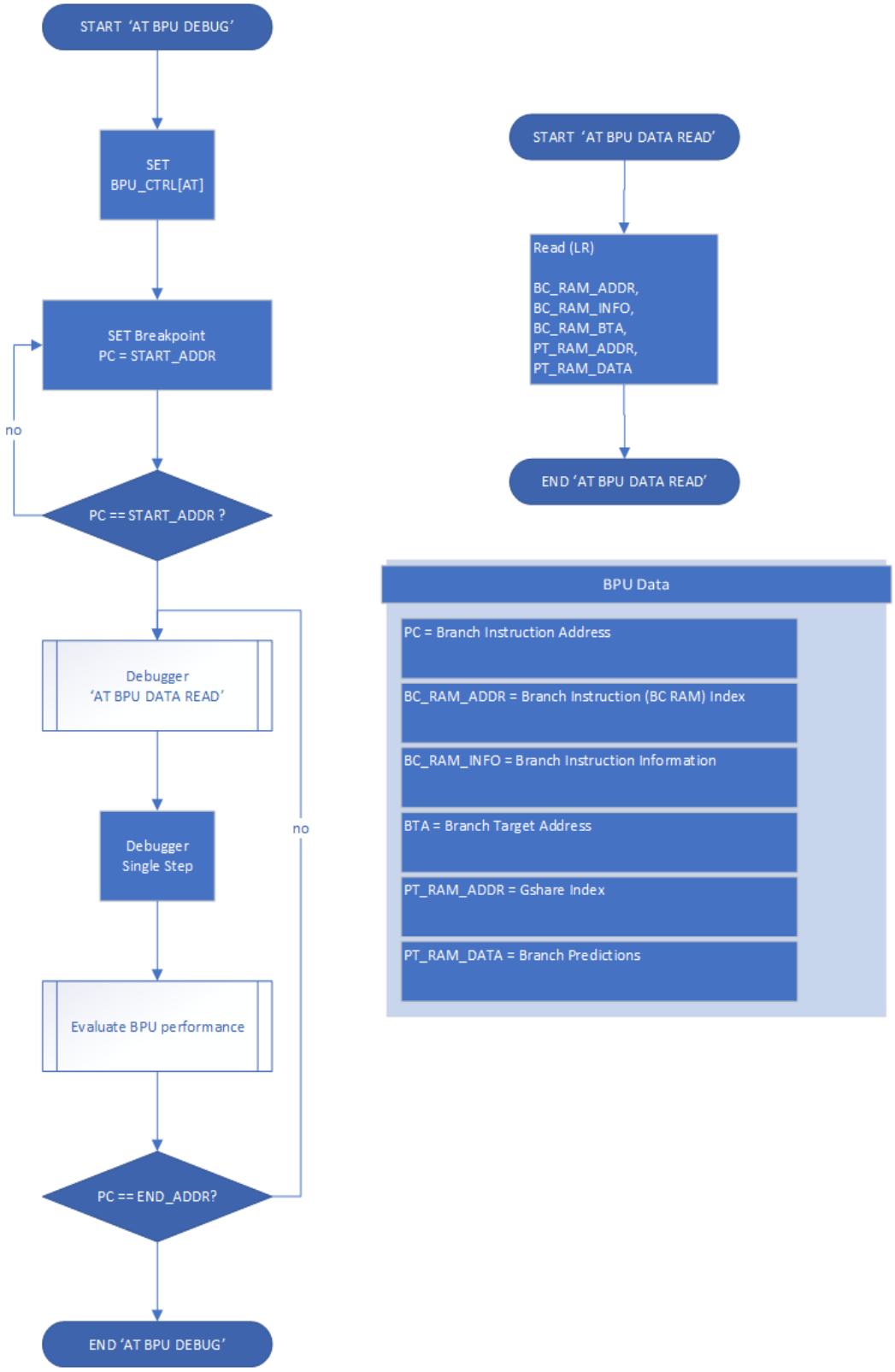
```
_bpu_int_excpn_hndl:
movr4,RAM_ST_ADDR;Access BPU control register
movr5,RAM_EN_ADDR;Access BPU control register

_bc_ram:
```

```
movr6, r4; program index to RAM_ST_ADDR
_bc_loop:
sr r6, [BPU_RAM_ADDR]; program BPU_RAM_ADDR
lr r7, [BC_RAM_INFO]; read BC RAM INFO
cmpr6, r5; program check index at end address beq.d _bc_loop; program loop to next RAM
address addr6, r6,0x4; program increment RAM address

bc_bta_ram:
movr6, r4; program index to RAM_ST_ADDR
_bc_bta_loop:
sr r6, [BPU_RAM_ADDR]; program BPU_RAM_ADDR
lr r7, [BC_RAM_BTA];  read BC RAM BTA
cmpr6, r5; program check index at end address beq.d _bc_bta_loop; program loop to next
RAM address addr6, r6,0x4; program increment RAM address

_pt_ram:
movr6, r4; program index to RAM_ST_ADDR
_pt_loop:
sr r6, [BPU_RAM_ADDR]; program BPU_RAM_ADDR
lr r7, [PT_RAM_DATA]; program read PT RAM DATA
cmpr6, r5; program check index at end address beq.d _pt_loop; program loop to next RAM
address addr6, r6,0x4; program increment RAM address
```

23.3.3.1 Read Access through BPU Debug Controlled Access Mode

Figure 23-8 showcase read access through BPU debug controlled access mode.

Figure 23-8 Read Access through BPU Debug Controlled Access Mode



24 External Host Debugging

24.1 External Host Debugging Introduction

If the configuration includes the interface to an external debug host, additional registers are included in the auxiliary register set. This auxiliary register set includes at least the DEBUG and DEBUGI register, and may also include other registers to support, for example, real-time tracing capabilities. For specific details of advanced debug capabilities, refer to the respective ARCv3 processor databook.

24.2 Host Debug Register Interface

Table 24-1 Host Debug Interface Auxiliary Registers

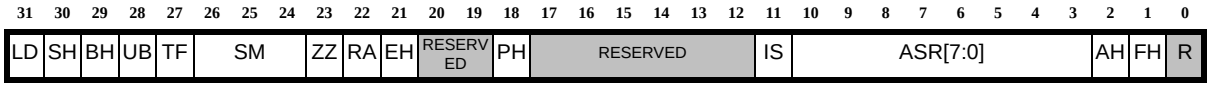
Address	Auxiliary Register Name	Description
0x5	Debug Register, DEBUG	Debug register

24.3 Debug Register, DEBUG

Address: 0x05

Access: RG

Figure 24-1 DEBUG Register



The debug register is standard in all ARCv3-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.

The debug register (DEBUG) contains the following bits:

Table 24-2 DEBUG Register Field Description

Field	Bit	Description
Force halt (FH)	[1]	FH is the approved method of stopping the ARCv3-based processor externally by the host. This bit does not have any affect if the ARCv3-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 *not* start the processor. The FH bit always returns 0 when it is read.
Actionpoint Halt Flag (AH)	[2]	The actionpoint halt flag (bit 2), of the DEBUG register, indicates that an actionpoint has halted the processor when the halt flag is set to 1. The AH flag is cleared when the halt flag (H) in the STATUS32 register (see Status Register, STATUS32) is cleared, for example, by restarting (see Starting) or single–stepping (see Single Instruction Stepping) the ARCv3-based processor. When actionpoints ause 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 24-2 DEBUG Register Field Description

Field	Bit	Description
Actionpoints Status Register field (ASR)	[10:3]	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]	Single instruction stepping is provided through the use of the IS bit. If the host sets the IS bit, the ARCv3 core executes one instruction.
Patch Halt (PH)	[18]	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]	The `EH` bit is set when the processor is halted by external logic. The `arc_halt_req_a` signal is asserted by external logic that wishes to halt the ARCv3-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 `arc_halt_req_a`, sets the `EH` bit, and asserts the `arc_halt_ack` acknowledgment signal. This bit is a status bit, and the host cannot write to this bit.
Reset applied (RA)	[22]	RA bit is used by the debug host to determine that a target reset has occurred. The RA bit is set to `1` on a hard reset of the CPU and can be written only by the host.

Table 24-2 DEBUG Register Field Description

Field	Bit	Description
Sleep state (ZZ)	[23]	ZZ bit indicates that the ARCv3-based processor is in the "sleep" state. Use the SLEEP instruction to force the ARCv3-based processor enter the sleep state. This bit is cleared whenever the ARCv3-based processor wakes from sleep state. This bit is a status bit, and the host cannot write to this bit.
Sleep Mode (SM)	[26:24]	SM field indicates the sleep mode of the processor. This field controls whether clocks for the internal timers and real-time clock are turned off during sleep state. For more information about the state of internal clocks during sleep state, see Table 24-3. This bit is a status bit, and the host cannot write to this bit.
Triple Fault (TF)	[27]	When this bit is set to 1, it indicates that the processor has halted due to a triple fault. A triple fault occurs when an exception occurs during the fetching of an exception vector for an EV_MachineCheck (double fault) exception.
User mode breakpoint (UB)	[28]	UB bit indicates that BRK is enabled in User mode. This bit is provided to allow an external debugger to debug User mode tasks. This bit is always set to 0 to ensure that a User mode task cannot stop the processor by executing a BRK instruction. If actionpoints are enabled, actionpoint halt in User mode requires that the DEBUG.UB bit is set.
Breakpoint halt (BH)	[29]	BH bit is set when 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 ARCv3-based processor from the halted state.
Self halt (SH)	[30]	SH indicates that the ARCv3-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 ARCv3-based processor is running or a single step has been executed.
Load pending bit (LD)	[31]	The host (see "The Host" on page 863) or the ARCv3-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 ARCv3-based processor. This bit is a status bit, and the host cannot write to this bit.

Table 24-3 lists the state of various internal clocks during various sleep modes.

Table 24-3 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

25 Debug Features

25.1 Debug Introduction

This chapter describes the debug features of the ARCv3-based processor series.These features include the following:

- Actionpoints
- SmaRT — Small Real-Time Trace Buffer
- Soft Reset
- Debug Interface

25.2 Actionpoints

The ARCv3 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 HS6x 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.

25.2.1 Actionpoint Auxiliary Registers

Actionpoints are programmed using a collection of control and status registers mapped into the auxiliary register space.

Table 25-1 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)
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)

Table 25-1 Auxiliary Registers for Actionpoints (Continued)

Auxiliary Register Address	Register Name	Description
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 6-3.

25.2.1.1 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

25.2.1.2 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.

25.2.1.3 Actionpoint Registers Reset State

Table 25-2 lists the values of the actionpoint registers when the ARCv3-based processor is reset.

Table 25-2 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
AP_WP_PC	0x00000000

25.2.2 Actionpoint Extensions to Debug Register, DEBUG, 0x05

The debug register is standard in all ARCv3-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 ARCv3-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).

25.2.3 Actionpoints Configuration Register, AP_BUILD

Address: 0x76

Access: R

The AP_BUIILD register (AP_BUILD) indicates the presence of the optional "Actionpoints" mechanism. The `AP_BUILD` register returns 0 when actionpoints are not available.

Figure 25-1 Actionpoint Build Register

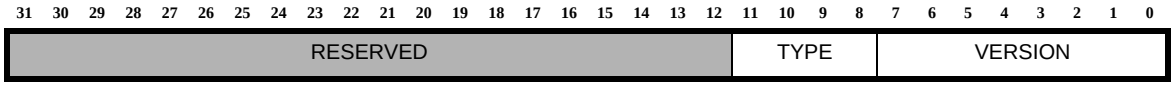


Table 25-3 Actionpoint Build Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Actionpoint Version ■ `0x6`: Current version of the actionpoint system
TYPE	[11:8]	Actionpoint Build Type Refers to the number of actionpoints selected in the ARCv3-based system: ■ `000x0`: 2 Actionpoints, full set of targets ■ `000x1`: 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 ■ `00011`, `0111-1111`: Reserved

25.2.4 Actionpoint Match Value, AP_AMV0

Address: 0x220

Access: RW

Figure 25-2 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.

25.2.5 Actionpoint Match Mask, AP_AMM0

Address: 0x221

Access: RW

Figure 25-3 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 0x00000000_12345000 to 0x00000000_12345FFF:

AP_AMV0 = 0x00000000_12345678

AP_AMM0 = 0x00000000_00000FFF

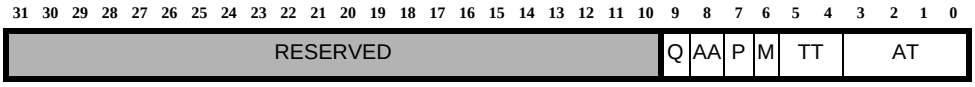
Depending on the setting of the Mode (M) bit in the Actionpoint Control register, the actionpoint triggers when the target value is within this range, or outside this range.

25.2.6 Actionpoint Control, AP_AC0

Address: 0x222

Access: RW

Figure 25-4 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.

25.2.6.1 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 25-4 lists the selections available.

Table 25-4 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.

25.2.6.2 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 25-5 lists the possible values: none (disabled), read, write, read or write.

Table 25-5 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.

25.2.6.3 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 25-6.

Table 25-6 Mode Field, M, Type

Mode (M)	Description
0x0	Trigger when in range
0x1	Trigger when outside range

25.2.6.4 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 25-7 Pair Field, P, Type

Pair (P)	Description
0	Normal unpaired operation
1	Pair this actionpoint with the next actionpoint

25.2.6.5 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 25-8 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.

25.2.6.6 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 25-9 Quad Type

Quad (Q)	Description
`0x0`	Normal operation
`0x1`	Quad this actionpoint with the next three actionpoints (modulo the number of actionpoints)

25.2.7 Actionpoint Match Value, AP_AMV1

Address: 0x223

Access: RW

Figure 25-5 Actionpoint 1 Match Value Register

See Actionpoint Match Value, AP_AMV0 register for field information.

25.2.8 Actionpoint Match Mask, AP_AMM1

Address: 0x224

Access: RW

Figure 25-6 Actionpoint 1 Match Mask Register

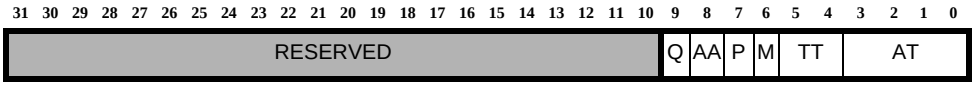
See Actionpoint Match Mask, AP_AMM0 register for field information.

25.2.9 Actionpoint Control, AP_AC1

Address: 0x225

Access: RW

Figure 25-7 Actionpoint 1 Control Register



See Actionpoint Control, AP_AC0 register for field information.

25.2.10 Actionpoint Match Value, AP_AMV2

Address: 0x226

Access: RW

Figure 25-8 Actionpoint 2 Match Value Register

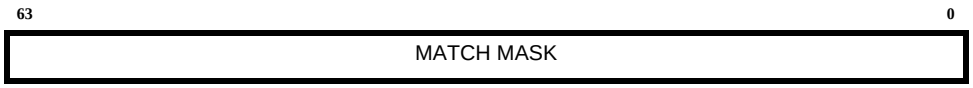
See Actionpoint Match Value, AP_AMV0 register for field information.

25.2.11 Actionpoint Match Mask, AP_AMM2

Address: 0x227

Access: RW

Figure 25-9 Actionpoint 2 Match Mask Register



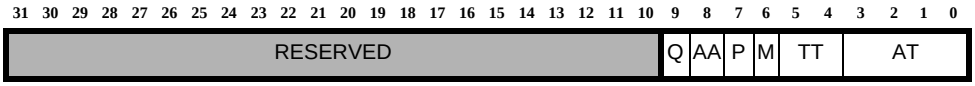
See Actionpoint Match Mask, AP_AMM0 register for field information.

25.2.12 Actionpoint Control, AP_AC2

Address: 0x228

Access: RW

Figure 25-10 Actionpoint 2 Control Register



See Actionpoint Control, AP_AC0 register for field information.

25.2.13 Actionpoint Match Value, AP_AMV3

Address: 0x229

Access: RW

Figure 25-11 Actionpoint 3 Match Value Register

See Actionpoint Match Value, AP_AMV0 register for field information.

25.2.14 Actionpoint Match Mask, AP_AMM3

Address: 0x22A

Access: RW

Figure 25-12 Actionpoint 3 Match Mask Register

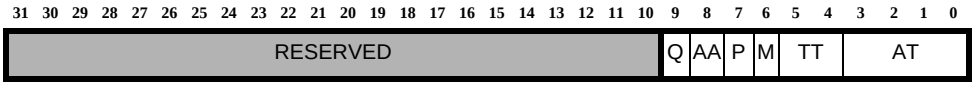
See Actionpoint Match Mask, AP_AMM0 register for field information.

25.2.15 Actionpoint Control, AP_AC3

Address: 0x22B

Access: RW

Figure 25-13 Actionpoint 3 Control Register



See Actionpoint Control, AP_AC0 register for field information.

25.2.16 Actionpoint Match Value, AP_AMV4

Address: 0x22C

Access: RW

Figure 25-14 Actionpoint 4 Match Value Register

See Actionpoint Match Value, AP_AMV0 register for field information.

25.2.17 Actionpoint Match Mask, AP_AMM4

Address: 0x22D

Access: RW

Figure 25-15 Actionpoint 4 Match Mask Register

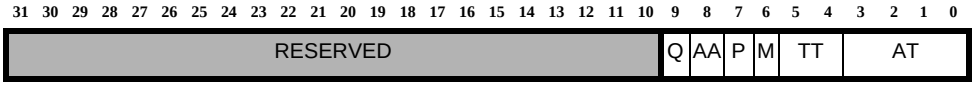
See Actionpoint Match Mask, AP_AMM0 register for field information.

25.2.18 Actionpoint Control, AP_AC4

Address: 0x22E

Access: RW

Figure 25-16 Actionpoint 4 Control Register



See Actionpoint Control, AP_AC0 register for field information.

25.2.19 Actionpoint Match Value, AP_AMV5

Address: 0x22F

Access: RW

Figure 25-17 Actionpoint 5 Match Value Register

See Actionpoint Match Value, AP_AMV0 register for field information.

25.2.20 Actionpoint Match Mask, AP_AMM5

Address: 0x230

Access: RW

Figure 25-18 Actionpoint 5 Match Mask Register

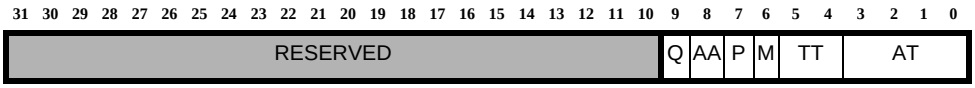
See Actionpoint Match Mask, AP_AMM0 register for field information.

25.2.21 Actionpoint Control, AP_AC5

Address: 0x231

Access: RW

Figure 25-19 Actionpoint 5 Control Register



See Actionpoint Control, AP_AC0 register for field information.

25.2.22 Actionpoint Match Value, AP_AMV6

Address: 0x232

Access: RW

Figure 25-20 Actionpoint 6 Match Value Register

See Actionpoint Match Value, AP_AMV0 register for field information.

25.2.23 Actionpoint Match Mask, AP_AMM6

Address: 0x233

Access: RW

Figure 25-21 Actionpoint 6 Match Mask Register

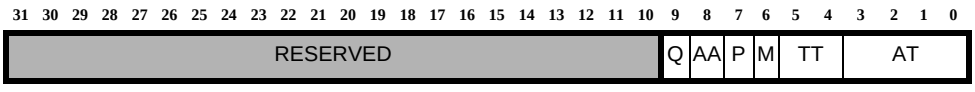
See Actionpoint Match Mask, AP_AMM0 register for field information.

25.2.24 Actionpoint Control, AP_AC6

Address: 0x234

Access: RW

Figure 25-22 Actionpoint 6 Control Register



See Actionpoint Control, AP_AC0 register for field information.

25.2.25 Actionpoint Match Value, AP_AMV7

Address: 0x235

Access: RW

Figure 25-23 Actionpoint 7 Match Value Register

See Actionpoint Match Value, AP_AMV0 register for field information.

25.2.26 Actionpoint Match Mask, AP_AMM7

Address: 0x236

Access: RW

Figure 25-24 Actionpoint 7 Match Mask Register

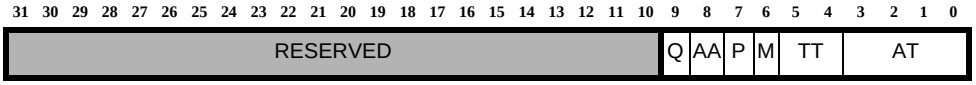
See Actionpoint Match Mask, AP_AMM0 register for field information.

25.2.27 Actionpoint Control, AP_AC7

Address: 0x237

Access: RW

Figure 25-25 Actionpoint 7 Control Register



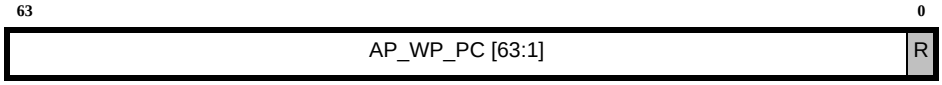
See Actionpoint Control, AP_AC0 register for field information.

25.2.28 Watchpoint Program Counter, AP_WP_PC

Address: 0x23F

Access: R

Figure 25-26 AP_WP_PC, Watchpoint Program Counter Register



AP_WP_PC is a read-only register.

An ARCv3-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.

25.3 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.

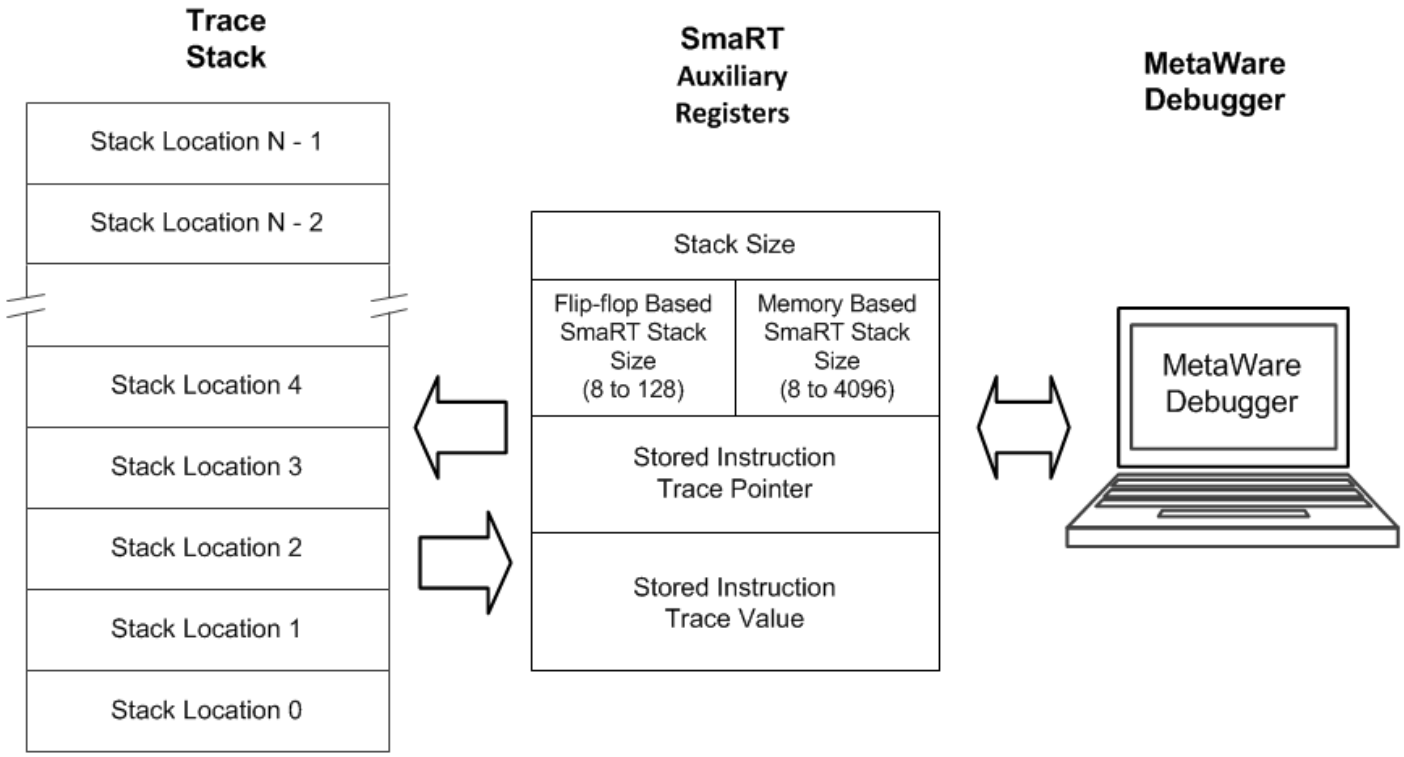
The MetaWare debugger enables SmaRT and displays the instruction trace history. The MetaWare debugger reads the saved instruction trace information via the JTAG port when the processor is halted.

25.3.1 Features

- Different stack sizes available — from 8 up to 4096 entries— to record program-flow changes.
- Any non-sequential instruction that is executed is stored including jumps, branches, interrupts and exceptions, and loops. Multiple branches to the same location (inner loop) are stored as a single entry to maximize stack usage.
- Power-saving features include gating the system clock and zeroing all inputs when not in trace mode.
- The MetaWare debugger enables SmaRT and displays trace information, eliminating the need for a separate, dedicated trace port.

25.3.2 Overview of SmaRT Operation

Figure 25-27 SmaRT Operation



- Trace Stack - stores source and destination addresses whenever a change in program flow is detected by the core. When an interrupt or exception is taken, the address of the vector is not recorded; the target address of the vector, the address of the first instruction of the handler, is recorded.

- SmaRT Auxiliary Registers:
  - SMART_BUILD — shows the configured size of the trace stack
  - SMART_CONTROL — controls SmaRT operation and points to a location in the trace stack
  - SMART_DATA — shows the value of a location in the trace stack (as defined in the SMART_CONTROL register)
- MetaWare Debugger — enables SmaRT and reconstructs the instruction trace

25.3.3 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

25.3.4 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 25-28 SMART_BUILD Value

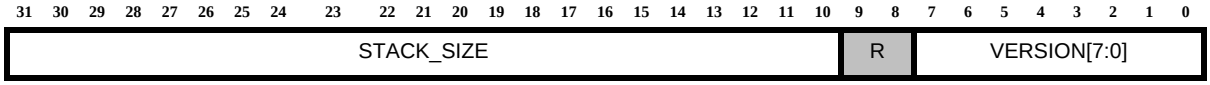


Table 25-10 lists the field descriptions:

Table 25-10 SMART_BUILD

Field	Bits	Description
VERSION	[7:0]	0x5: Version of SmaRT component.
R	[9:8]	Reserved Returns zero when read.

Table 25-10 SMART_BUILD

Field	Bits	Description
STACK_SIZE	[31:10]	Trace stack size Indicates the stack size and allows the debugger to identify the capabilities of the implementation for display purposes. Possible values are from 8 up to 4096. Values above 128 are only available for the SRAM-based implementation.

25.3.5 SmaRT Control Register, SMART_CONTROL

Address: 0x700

Access: RW

The SMART_CONTROL register has the following contents:

Figure 25-29 SMART_CONTROL Register

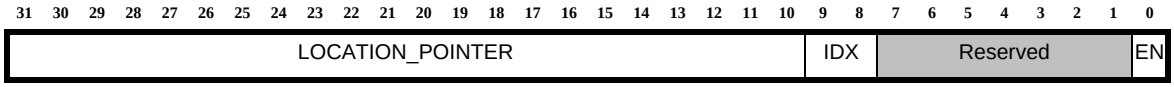


Table 25-11 lists the field descriptions:

Table 25-11 SMART_CONTROL Register

Field	Bits	Description
EN	[0]	SmaRT enable Enables the trace mechanism. ■ `0x0`: Trace Disabled. ■ `0x1`: Trace Enabled.
Reserved	[7:1]	Reserved Returns zero when reading. Ignored when writing.
IDX[1:0]	[9:8]	Location index Controls which value is returned in the SMART_DATA register. ■ `0x0`: SRC_ADDR value ■ `0x1`: DEST_ADDR value ■ `0x2`: FLAGS value ■ `0x3`: Reserved
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.

25.3.6 SmaRT Data Register, SMART_DATA

Address: 0x701

Access: R

The SMART_DATA read-only register has the following contents:

Figure 25-30 SMART_DATA Register

Table 25-12 lists the field descriptions:

Table 25-12 SMART_DATA Register

Field	Bits	Description
LOCATION_VALUE	[63:0]	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.

25.3.7 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 25-31 SRC_ADDR Value

Table 25-13 lists the field descriptions:

25.3.8 DEST_ADDR Value, Index 1

Table 25-13 SRC_ADDR Value

Field	Bits	Description
LOCATION_VALUE	[63:0]	Stored instruction trace stack location value The value of the source address in the stack contained at the location given in the LOCATION_POINTER field of the SMART_CONTROL register.

The DEST_ADDR value is read from the SMART_DATA register when IDX field in the SMART_CONTROL register is set to 1.

Figure 25-32 DEST_ADDR Value

Table 25-14 lists the field descriptions:

Table 25-14 DEST_ADDR Register

Field	Bits	Description
LOCATION_VALUE	[63:0]	Stored instruction-trace stack-location value The value of the destination address in the stack contained at the location given in the LOCATION_POINTER field of the SMART_CONTROL register.

25.3.9 FLAGS Value, Index 2

The FLAGS value is read from the SMART_DATA register when the IDX field in the SMART_CONTROL register is set to 2. The field descriptions are as follows:

Figure 25-33 FLAGS Value Register

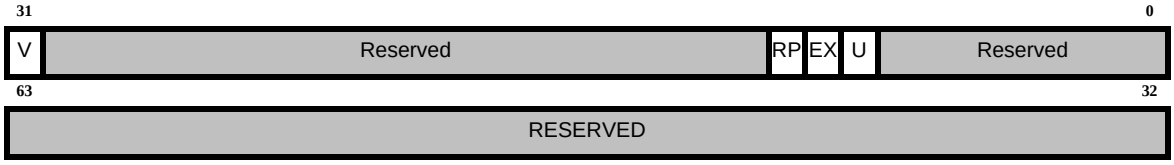


Table 25-15 lists the field descriptions:

Table 25-15 FLAGS Value Register

Field	Bits	Description
Reserved	[7:0]	Reserved Returns zero when read.
U	[8]	User mode Set if the processor is in User mode after program-flow change.
EX	[9]	Exception Indicates that the program-flow change is a result of an exception or interrupt.
RP	[10]	Repeat Used to indicate that the same program-flow change occurred more than once.
Reserved	[30:11]	Reserved Returns zero when read.
V	[31]	Valid Indicates that the value in the stack is valid.
Reserved	[63:32]	Reserved Returns zero when read.

25.3.9.1 Reserved Value, Index 3

Zero is read from the SMART_DATA register when IDX field in the SMART_CONTROL register is set to 3.

25.4 Soft Reset

Soft reset provides a mechanism to recover from system malfunctioning. Other system-level monitoring mechanisms can be deployed to initiate a soft reset procedure. The effect of a soft reset is similar to that of a normal ("hard") reset, except that part of the state is not affected, or affected differently:

- The shared cache and memory (SCM) is not cleared, even if CFG_SCM_RESET is configured.
- All debug registers and memories retain their state from just before the soft reset activation.

25.4.1 Soft Reset Capability for Debug Host

The soft reset capability is accessed by the host using bit 31 of the DEBUGI auxiliary register. Writing 1 to this bit triggers a soft reset of the entire cluster. This write is intercepted by the debug interface, which therefore does not attempt to write to DEBUGI using a debug instruction. This is important, as the core may be unresponsive, and in that condition the debug interface cannot issue a debug instruction.

25.4.2 Interrupt and Register Bank Debug Register, DEBUGI

Address: 0x0F

Access: RW

Reset: 0x0000_0000

Figure 25-34 DEBUGI Register

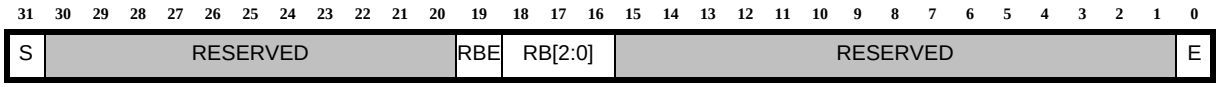


Table 25-16 DEBUGI Register Field Description

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.
RB	[18:16]	Register Bank (present with `RGF_NUM_BANKS > 1`) Pre-empts STATUS32.RB for all instructions originating from the debug interface when DEBUGI.RBE is set. When DEBUGI.RBE is not set, the existing value STATUS32.RB is used where RBE = bit 19. If `RGF_NUM_BANKS {== 1, < 2, < 4}`, the {RB, RB[2:1], RB[2]} field is read as zero and ignored on write. 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 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.

Table 25-16 DEBUGI Register Field Description

Field	Bit	Description
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.
S	[31]	Writing the value 1 to the DEBUGI.S triggers a soft reset to the entire cluster.

25.4.3 Core State After Soft Reset

The state of the ARCv3 cores after a soft reset is similar to their state after a regular ("hard") reset, with the following exceptions:

- The L1 caches retain their state, that is they are not cleared
- The L2 TLB retain its state
- The instruction and data CCMs retain their state (even if configured with option to initialize on hard reset)
- The soft reset auxiliary registers capture the state of the core at the time the soft reset is applied

25.4.4 Soft Reset Auxiliary Registers

Upon a soft reset, the core saves the architectural PC and some of its state in auxiliary registers.

25.4.5 CLN Soft Reset Cores, CLN_SOFTRESET_CORES

CLNR Address: 0xB00

Access: RW

Figure 25-35 CLN_SOFTRESET_DEV Register

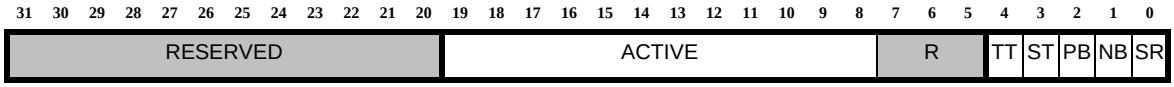


Table 25-17 CLN_SOFTRESET_CORES

Field name	Range	Access	Description
SR	[0:0]	R; IOW	This bit must be tested as part of the boot procedure. If set to 1, this indicates that the cluster soft reset was activated, and therefore that some state is not cleared. If this bit is 0, the system is fully cleared as specified by the @reset tags in this register. @reset=0; @softreset=1.
NOC_BUSY (NB)	[1:1}	R; IOW	If this bit is 1, at least one NoC master port has not completed all its outstanding transactions when the soft reset was activated. @reset=0; @softreset=(as described)
PER_BUSY(PB)	[2:2}	R; IOW	If 1, at least one configured PER master port had not completed all its outstanding transactions when the soft reset was activated. When no PER master ports are configured this bit is always 0. @reset=0; @softreset=(as described)
SCU_TRC(ST)	[3:3]	R; IOW	Copy of CLN_SCU_TRC_CTRL.E at the time that soft reset was activated.
TXC_TRC(TT)	[4:4]	R; IOW	Copy of CLN_TXC_TRC_CTRL.E at the time that soft reset was activated.
R	[7:5]	RAZ; IOW	Reserved.

Table 25-17 CLN_SOFTRESET_CORES

Field name	Range	Access	Description
ACTIVE	[19:8]	R; IOW	ACTIVE[*k*] is 1 if ARC core k had at least one memory transaction pending when the soft reset was activated; and it is 0 otherwise; or if core k does not exist. The ACTIVE bits are always cleared on a normal (hard) reset. @reset=0; @softreset=(as described)
RESERVED	[31:20]	RAZ; IOW	Reserved.

25.4.6 CLN Soft Reset Device Ports, CLN_SOFTRESET_DEV

CLNR Address: 0xB01

Access: RW

Figure 25-36 CLN_SOFTRESET_DEV Register

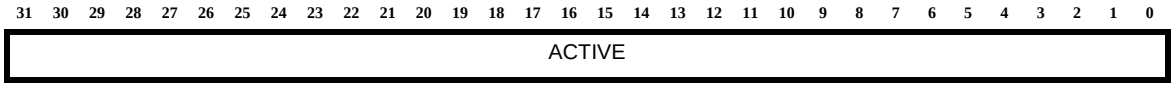


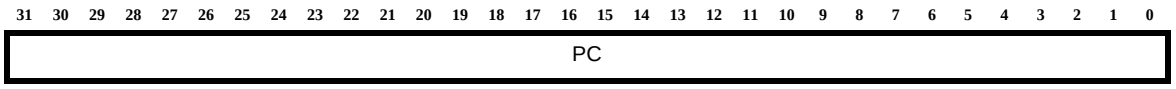
Table 25-18 CLN_SOFTRESET_DEV

Field name	Range	Access	Description
ACTIVE	[31:0]	R; IOW	ACTIVE[k] is 1 if device port *k* had at least one memory transaction pending when the soft reset was activated, and it is 0 otherwise or if DEV *k* does not exist. The ACTIVE bits are always cleared on a normal ("hard") reset. @reset=0,@softreset=(as described).

25.4.7 Soft Reset PC, SOFT_RESET_PC

Core Aux Address:	0x3A0
Access:	RG
Size	ADDR_SIZE

Figure 25-37 SOFT_RESET_PC Register

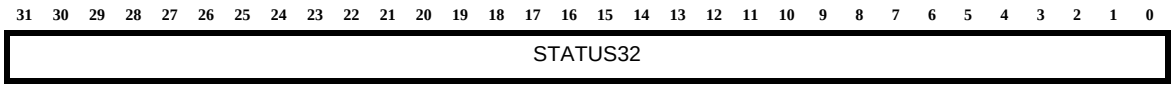


Program Counter of the last instruction committed before the soft reset exception.

25.4.8 Soft Reset STATUS32, SOFT_RESET_STATUS32

Core Aux Address:	0x3A1
Access:	RG
Size	32

Figure 25-38 SOFT_RESET_STATUS32 Register

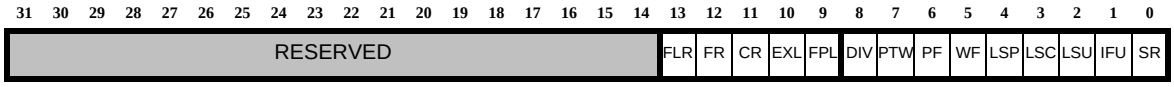


The contents of the STATUS32 auxiliary register is saved to this register upon a soft reset.

25.4.9 Soft Reset State, SOFT_RESET_STATE

Core Aux Address:	0x3A2
Access:	RG
Size	32
Hard Reset	0x0000_0000

Figure 25-39 SOFT_RESET_STATUS32 Register



The contents of the core state is saved to this register upon a soft reset.

Table 25-19 SOFT_RESET_STATE

Field name	Range	Description
Soft Reset (SR)	[0]	This bit is asserted upon a soft reset. This bit is reset upon a regular (hard) reset. This bit shall be tested as part of the boot procedure to differentiate between a soft (debug) reset or a regular reset
IFU busy(IFU)	[1]	This bit is asserted when a soft reset is applied while there are outstanding IFU memory transactions
Load/store uncached (LSU)	[2]	This bit is asserted when a soft reset is applied while there are outstanding load/store uncached memory transactions
Load/store cached (LSC)	[3]	This bit is asserted when a soft reset is applied while there are outstanding load/store cached memory transactions
Load/store private peripheral (LSP)	[4]	This bit is asserted when a soft reset is applied while there are outstanding load/store private peripheral transactions
Write streaming (WS)	[5]	This bit is asserted when a soft reset is applied while there are outstanding streaming writes from the Hardware prefetcher
Hardware prefetch busy (PF)	[6]	This bit is asserted when a soft reset is applied while there are outstanding (read) prefetch transactions
Page table walker busy (PTW)	[7]	This bit is asserted when a soft reset is applied while there are outstanding MMU page-table-walker transactions
Divider busy (DIV)	[8]	This bit is asserted when a soft reset is applied while there are outstanding integer divide instructions
FPU busy (FPL)	[9]	This bit is asserted when a soft reset is applied while there are outstanding floating-point instructions

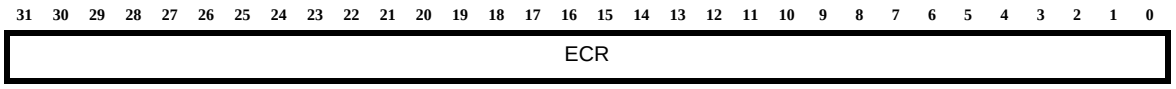
Table 25-19 SOFT_RESET_STATE

Field name	Range	Description
EXU busy (EXL)	[10]	This bit is asserted when a soft reset is applied while the Execution Unit is not idle.
Pending core register retirement (CR)	[11]	This bit is asserted when a soft reset is applied while there are pending retirements to the general-purpose register file.
Pending FPU register retirement (FR)	[12]	This bit is asserted when a soft reset is applied while there are pending retirements to the floating-point register file.
Pending Flag retirement (FLR)	[13]	This bit is asserted when a soft reset is applied while there are pending retirements to the floating-point register file.

25.4.10 Soft Reset Saved ECR, SOFT_RESET_ECR

Core Aux Address:	0x3A3
Access:	RG
Size	32

Figure 25-40 SOFT_RESET_ECR Register

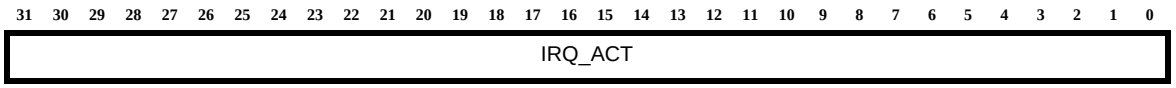


The contents of the ECR auxiliary register is saved to this register upon a soft reset.

25.4.11 Soft Reset Save Active Interrupt Register, SOFT_RESET_IRQ_ACT

Core Aux Address:	0x3A4
Access:	RG
Size	32

Figure 25-41 SOFT_RESET_IRQ_ACT Register



The contents of the AUX_IRQ_ACT auxiliary register is saved to this register upon a soft reset.

25.5 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.

25.5.1 ARC Debug-Access Registers

Table 25-20 lists the names, addresses, and reset values of the ARC debug-access registers.

Table 25-20 Debug Register Addresses

Auxiliary Address	Address over Debug Port	Reset value	Name
0xFFFF_0000	0x0000	0x24	"Debug Status Register, DB_STATUS"
0xFFFF_0004	0x0004	0x0	"Debug Command Register, DB_CMD"
0xFFFF_0008	0x0008	0x0	"Debug Address Register, DB_ADDR"
0xFFFF_0018	0x0018	0x0	"Debug Address Register, DB_ADDR_HI"
0xFFFF_000C	0x000C	0x0	"Debug Data Register, DB_DATA"
0xFFFF_0014	0x0014	0x0	"Debug Data Register, DB_DATA_HI"
0xFFFF_0010	0x0010	0x0	"Debug Reset Register, DB_RESET"

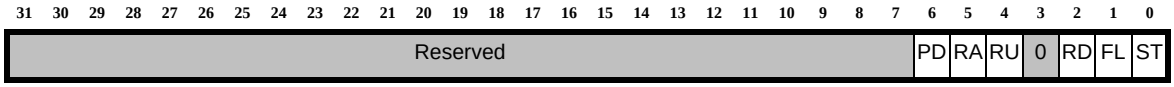
25.5.2 Debug Status Register, DB_STATUS

Address: 0xFFFF_0000

Access: RW

The DB_STATUS register provides information on all DEBUG operations. Writing to this register initiates the debug operation defined in DB_CMD. Read access by DB_STATUS and Write access by DB_EXEC.

Figure 25-42 DB_STATUS Register



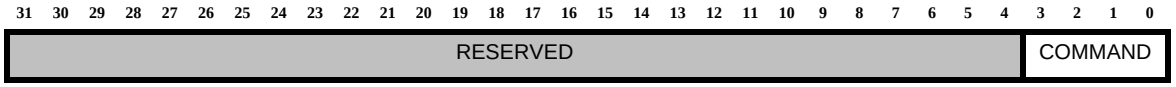
- Bit 0 - The stalled (ST) flag indicates that the debug interface is busy.
- Bit 1 - When true, the failure (FL) flag indicates that the debug operation has failed.
- Bit 2 - The ready (RD) flag indicates whether the debug interface is available to accept another transaction command.
- Bit 3 - Always zero.
- Bit 4 - The run (RU) flag is set to one when the ARC HS6x processor is running.
- Bit 5 - The Reset Applied (RA) flag is used by the debug host to determine that a target reset has occurred.
- Bit 6 - The power down (PD) flag indicates any transaction failures due to powering down of the core.
- Bits 31 to 6 - Reserved.

25.5.3 Debug Command Register, DB_CMD

Address: 0xFFFF_0004

Access: RW

Figure 25-43 DB_CMD Register



The DB_CMD register defines the debug command (read/write to aux/core/mem). All non-enumerated values result in a NOP instruction.

Table 25-21 DB_CMD Field Description

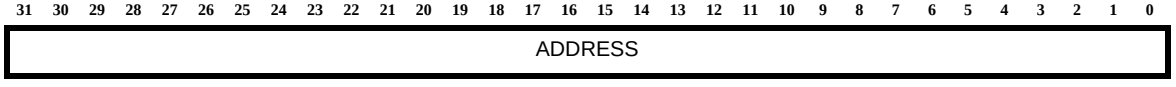
Field	Description
0x0	Write to Memory Location (CMD_WR_MEM)
0x1	Write to Core Register (CMD_WR_CORE)
0x2	Write to Auxiliary Register (CMD_WR_AUX)
0x3	Results in a NOP instruction
0x4	Read Memory Location (CMD_RD_MEM)
0x5	Read Core Register (CMD_RD_CORE)
0x6	Read Auxiliary Register (CMD_RD_AUX)
0x8	Write Memory Register (CMD_WR_MEM64)
0xC	Read Memory Register (CMD_RD_MEM64)

25.5.4 Debug Address Register, DB_ADDR

Address: 0xFFFF_0008

Access: RW

Figure 25-44 DB_ADDR Register



The DB_ADDR register (lower 32 bits) specifies:

- A core register when DB_CMD is CMD_RD_CORE or CMD_WR_CORE
- An auxiliary register when DB_CMD is CMD_RD_AUX or CMD_WR_AUX

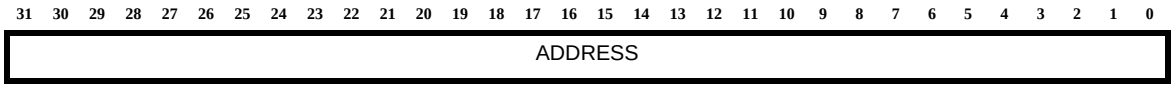
- The physical memory address when DB_CMD is CMD_RD_MEM or CMD_WR_MEM

25.5.5 Debug Address Register, DB_ADDR_HI

Address: 0xFFFF_0018

Access: RW

Figure 25-45 DB_ADDR_HI Register



The DB_ADDR_HI register specifies upper 32 bits of:

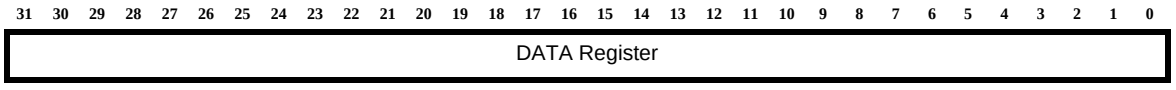
- A core register, and ignored when DB_CMD is CMD_RD_CORE or CMD_WR_CORE
- An auxiliary register when DB_CMD is CMD_RD_AUX or CMD_WR_AUX
- The physical memory address when DB_CMD is CMD_RD_MEM or CMD_WR_MEM

25.5.6 Debug Data Register, DB_DATA

Address: 0xFFFF_000C

Access: RW

Figure 25-46 DB_DATA Register



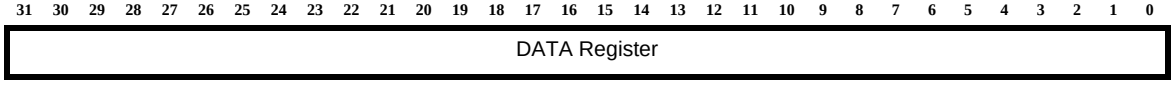
This register is loaded with the lower 32 bits of write data prior to a writing to DB_EXEC, when DB_CMD is CMD_WR. It contains the lower 32 bits of data returned by a successful debug command, when DB_CMD is CMD_RD.

25.5.7 Debug Data Register, DB_DATA_HI

Address: 0xFFFF_0014

Access: Read/Write

Figure 25-47 DB_DATA_HI Register



This register is loaded with the upper 32 bits of write data prior to a writing to DB_EXEC, when DB_CMD is CMD_WR_CORE, CMD_WR_AUX, or CMD_WR_MEM64. It contains the upper 32 bits of data returned by

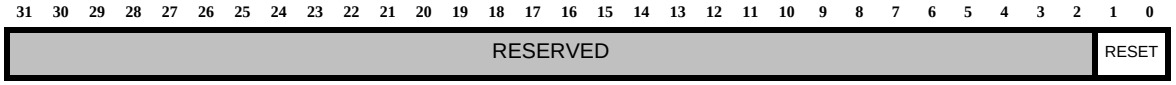
a successful debug command, when DB_CMD is DB_CMD is CMD_RD_CORE, CMD_RD_AUX, or CMD_RD_MEM64.

25.5.8 Debug Reset Register, DB_RESET

Address: 0xFFFF_0010

Access: Read/Write

Figure 25-48 DB_RESET Register



Writing any non-zero value to the DB_RESET register triggers a system reset request.

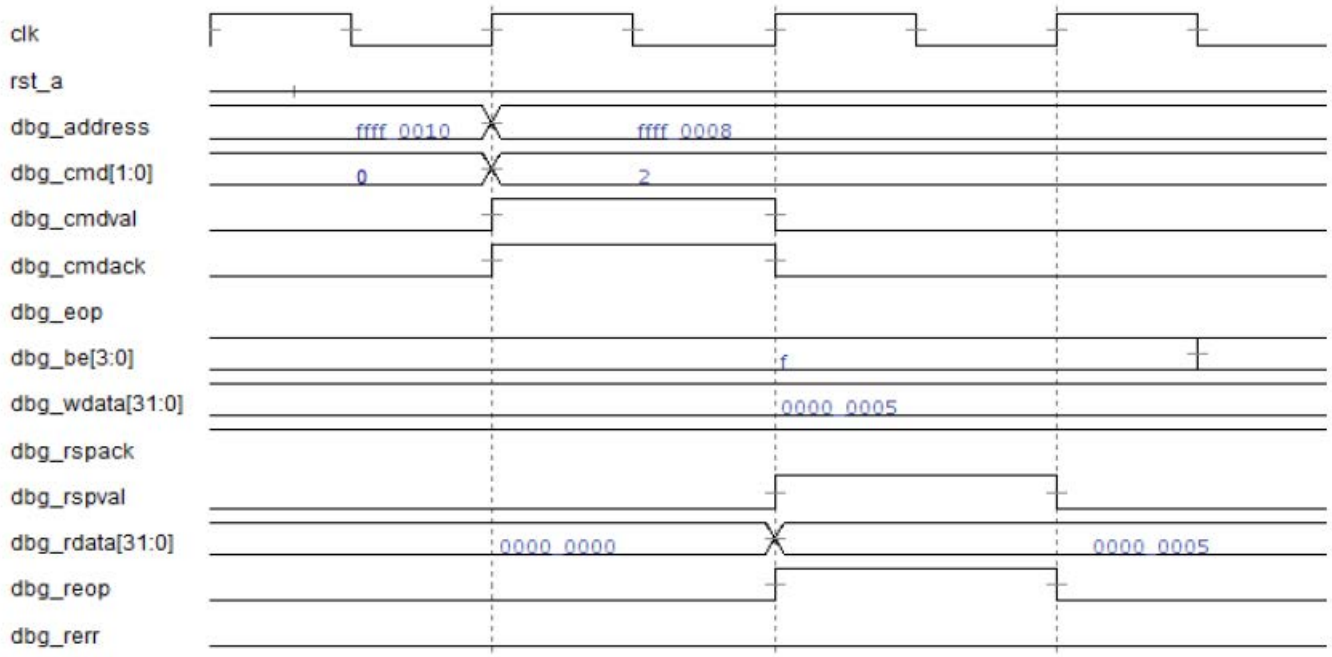
25.5.8.1 Initiating a Basic Debug Operation

Note This register is implemented only if the core is configured without a JTAG interface, and used to allow a software debug host reset the core by the debug interface.

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 25-49 Debug Interface Timing Diagram



26 Performance Counters

26.1 Performance Counters Introduction

The ARCv3-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_CTRLregister to control the dynamic power consumption.

26.2 Performance Counters Registers

The following registers provided for interacting with the performance-counter block.

Table 26-1 Performance Auxiliary Registers

Address	Auxiliary Register Name	Description
0x250	Count-Value Registers, PCT_COUNTL	Lower order current-count value
0x252	Snapshot-Value Registers, PCT_SNAPL	Lower 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
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 26-2 are provided for interaction with the countable conditions block.

Table 26-2 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

Table 26-2 Countable-Conditions Registers

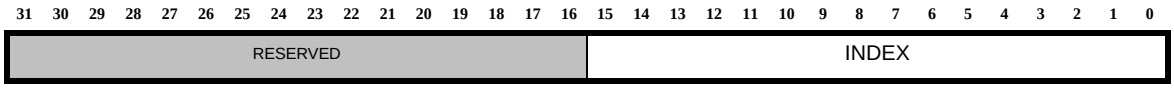
Address	Name	Description
0x241	Countable Conditions Name0 Register, CC_NAME0	Read first four characters of condition name, set by CC_INDEX
0x242	Countable Conditions Name1 Register, CC_NAME1	Read second four characters of condition name, set by CC_INDEX

26.2.1 Countable Conditions Index Register, CC_INDEX

Address: 0x240

Access: RW

Figure 26-1 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 26-3 CC_INDEX Field Description

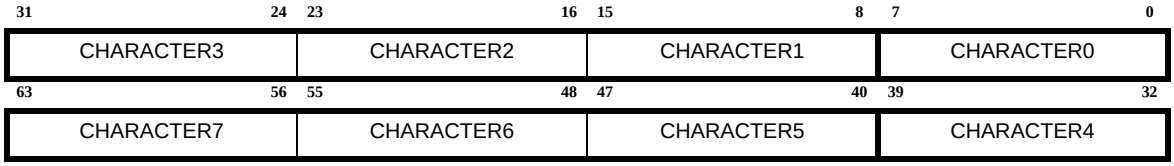
Field	Bit	Description
INDEX	[15:0]	Index of the countable condition whose name is returned through the registers `CC_NAME0` and `CC_NAME1`. The index values start at zero.

26.2.2 Countable Conditions Name0 Register, CC_NAME0

Address: 0x241

Access: R

Figure 26-2 CC_NAME0 Register



This register provides the [7:0] characters of the countable condition name corresponding to the index in 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 16 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 26-4 CC_NAME0 Field Description

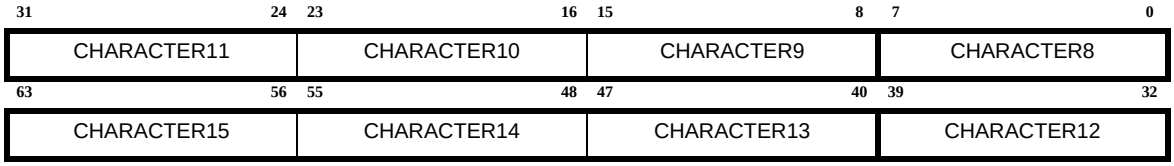
Field	Bit	Description
CHARACTER0	[7:0]	Character 0
CHARACTER1	[15:8]	Character 1
CHARACTER2	[23:16]	Character 2
CHARACTER3	[31:24]	Character 3
CHARACTER4	[39:32]	Character 4
CHARACTER5	[47:40]	Character 5
CHARACTER6	[55:48]	Character 6
CHARACTER7	[63:56]	Character 7

26.2.3 Countable Conditions Name1 Register, CC_NAME1

Address: 0x242

Access: R

Figure 26-3 CC_NAME1 Register



This register provides the [15:8] characters of the countable condition name corresponding to the index in 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 16 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 26-5 CC_NAME1 Field Description

Field	Bit	Description
CHARACTER8	[7:0]	Character 8
CHARACTER9	[15:8]	Character 9
CHARACTER10	[23:16]	Character 10
CHARACTER11	[31:24]	Character 11
CHARACTER12	[39:32]	Character 12
CHARACTER13	[47:40]	Character 13
CHARACTER14	[55:48]	Character 14
CHARACTER15	[63:56]	Character 15

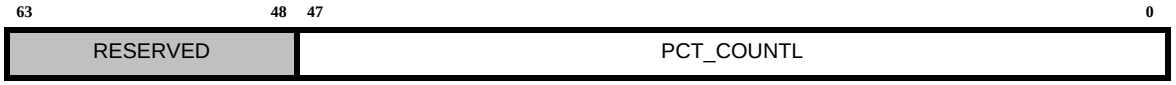
26.2.4 Count-Value Registers, PCT_COUNTL

Address: 0x250

Access: RW

Each count value is a 48-bit unsigned integer. The `PCT_INDEX` register controls access to the counter:

Figure 26-4 PCT_COUNTL Register



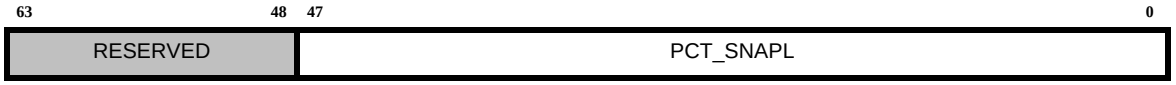
Count values wrap to zero when the maximum count value is reached. If the register is written with a specific value, the counter starts from this value.

26.2.5 Snapshot-Value Registers, PCT_SNAPL

Address: 0x252

Access: RW

Figure 26-5 PCT_SNAPL Register



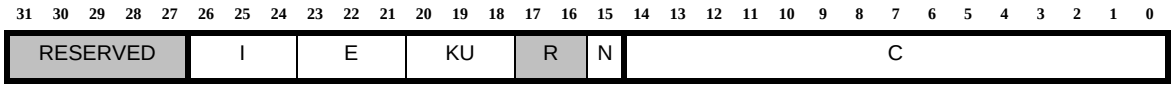
The snapshot feature allows simultaneous capture of all count values without halting the ongoing counting. The count value is a 48-bit unsigned integer, presented in the 64-bit register PCT_SNAPL. The `PCT_INDEX` register controls access to the counter snapshot. Count values from each counter are copied into the associated snapshot registers when a snapshot is triggered through the `PCT_CONTROL` register. Index selection has no effect on snapshots.

26.2.6 Configuration Register, PCT_CONFIG

Address: 0x254

Access: RW

Figure 26-6 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 26-6 PCT_CONFIG Field Description

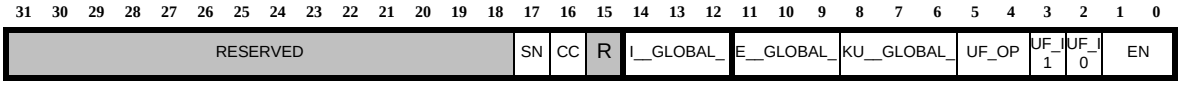
Field	Bit	Description
C	[14:0]	Countable condition index number
N	[15]	Invert condition before use
KU	[20:18]	■ 0x0: count always ■ 0x1: count only when in user mode ■ 0x2: count only when in kernel mode ■ 0x3 to 0x7: reserved
E	[23:21]	■ 0x0: count always ■ 0x1: count only when an exception is active ■ 0x2: count only when an exception is not active ■ 0x3 to 0x7: reserved
I	[26:24]	■ 0x0: count always ■ 0x1: count only when an interrupt is taken ■ 0x2: count only when an interrupt is not taken ■ 0x3 to 0x7: reserved

26.2.7 Control Register, PCT_CONTROL

Address: 0x255

Access: RW

Figure 26-7 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 26-7 PCT_CONTROL Field Description

Field	Bit	Description
EN	[1:0]	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]	■ `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])
UF_I1	[3]	■ `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])

Table 26-7 PCT_CONTROL Field Description

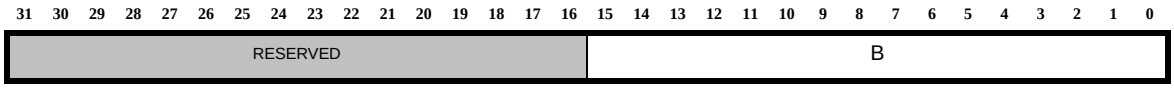
Field	Bit	Description
UF_OP	[5:4]	User flag operand ■ `0x0`: enable counting when (`uflag0 XOR UF_I0`) AND (`uflag1 XOR UF_I1`) `==1` ■ `0x1`: enable counting when (`uflag0 XOR UF_I0`) OR (`uflag1 XOR UF_I1`) `==1` ■ `0x2`: enable counting when `(uflag0 XOR UF_I0)==1` ■ `0x3`: enable counting when `(uflag1 XOR UF_I1)==1`
KU__GLOBAL_	[8:6]	■ `0x0`: count always ■ `0x1`: count only when in user mode ■ `0x2`: count only when in kernel mode ■ `0x3-0x7`: reserved
E__GLOBAL_	[11:9]	■ `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]	■ `0x0`: count always ■ `0x1`: count only when an interrupt is active ■ `0x2`: count only when an interrupt is NOT active ■ `0x3-0x7`: reserved
CC	[16]	Clear counts. When this bit is written with `1`, all counters are cleared to zero. Write-only – returns zero on read
SN	[17]	Snapshot. When this bit is written with `1`, all count values are copied into the snapshot registers. Write-only – returns zero on read

26.2.8 Index-Select Register, PCT_INDEX

Address: 0x256

Access: RW

Figure 26-8 PCT_INDEX Register



This register is used to select the counters to be accessed through count, snapshot, and configuration registers.

Table 26-8 PCT_INDEX Field Description

Field	Bit	Description
B[15:0]	[15:0]	Counter number. Selects the counters to be accessed through count, snapshot, and configuration registers. Counter operation is not affected by index selection.

26.2.9 Address-Range Registers, PCT_RANGEL

Address: 0x259

Access: RW

Figure 26-9 PCT_RANGEL Register

Address range registers can be used to enable counting only when the program counter is within a specific address range. Counting is enabled when `(PCT_RANGEL <= PC)` and either `(PC < PCT_RANGEH)` or `(PCT_RANGEH == 0).`

This register defaults to `0x0` on reset.

26.2.10 Address-Range Registers, PCT_RANGEH

Address: 0x25A

Access: RW

Figure 26-10 PCT_RANGEL Register

Address range registers can be used to enable counting only when the program counter is within a specific address range. Counting is enabled when `(PCT_RANGEL <= PC)` and either `(PC < PCT_RANGEH)` or `(PCT_RANGEH == 0)`.

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 0xfffffffe+2=0x0).

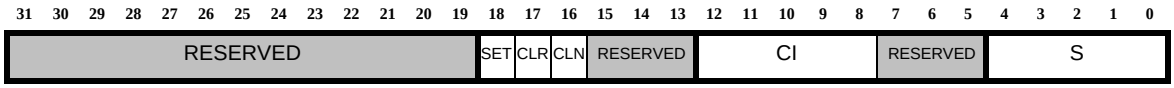
This register defaults to `0x0` on reset.

26.2.11 User-Flag Register, PCT_UFLAGS

Address: 0x25B

Access: RW

Figure 26-11 PCT_UFLAGS Register



User-settable flags (uflag*x*) are provided for instrumenting a specific program. Table 26-9 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 26-1 describes a few example.

Example 26-1 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 26-9 PCT_UFLAGS Field Description

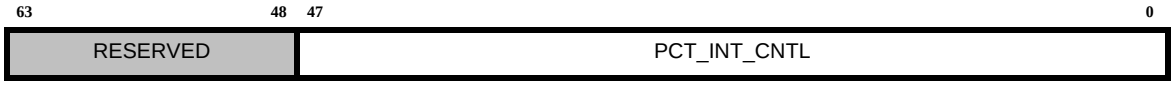
Field	Bit	Description
S	[4:0]	Set-flag index
CI	[12:8]	Clear-flag index
CLN	[16]	Clear all flags
CLR	[17]	Clear flag specified by CI field
SET	[18]	Set flag specified by SI field, after clearing operations have taken place

26.2.12 Interrupt Trigger Value, PCT_INT_CNTL

Address: 0x25C

Access: RW

Figure 26-12 PCT_INT_CNTL Register



Performance counters can be configured to trigger interrupt when a count is reached. Each count value is an unsigned 48-bit integer. The counter to be accessed is controlled by the `PCT_INDEX` register.

Count values wrap to zero when the maximum count value is reached.

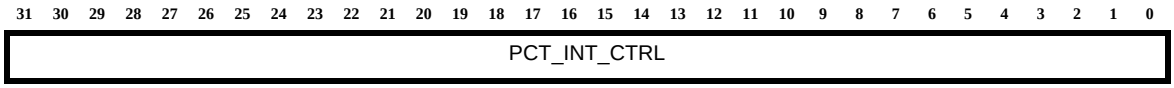
The enable bit corresponding to the counter must also be set for the interrupt to be triggered.

26.2.13 Interrupt Control Value, PCT_INT_CTRL

Address: 0x25E

Access: RW

Figure 26-13 PCT_INT_CTRL Register



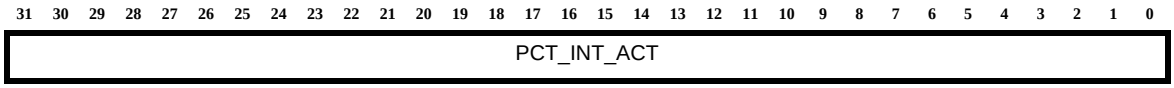
Each bit in this register is an interrupt enable bit for performance counter *X*, where *X* is the counter number.

26.2.14 Interrupt Active, PCT_INT_ACT

Address: 0x25F

Access: RW

Figure 26-14 PCT_INT_ACT Register



This register indicates the interrupts that are active (triggered).

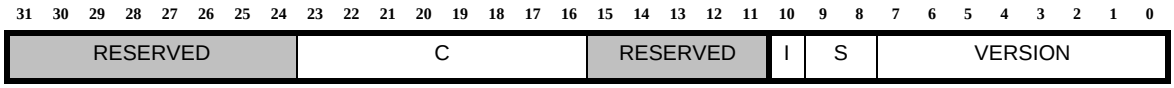
Field	Bit	Description
PCT_INT_ACT	[31:0]	Indicates the counters that have triggered an interrupt. For example, if counter *X* has triggered an interrupt, the `PCT_INT_ACT[`*X*`]` bit is set to `1`. Software must write a `1` to the appropriate bit in the `PCT_INT_ACT` register to clear the corresponding 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`.

26.2.15 Performance Counter Build-Configuration Register, PCT_BUILD

Address: 0xF5

Access: R

Figure 26-15 PCT_BUILD Register



The performance-counter build-configuration register contains the fields shown in Table 26-10 on page 1094.

Table 26-10 PCT_BUILD Field Description

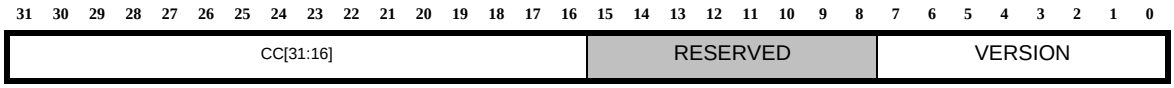
Field	Bit	Description
VERSION	[7:0]	Version number. All non-enumerated values are reserved. ■ `0x0`: not present ■ `0x3`: current version of the performance counter component
S	[9:8]	Counter size: All non-enumerated values are reserved. ■ `0x0`: 32 bit ■ `0x1`: 48 bit (default) ■ `0x2`: 64 bit All other 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
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

26.2.16 Countable Conditions Build Configuration Register, CC_BUILD

Address: 0xF6

Access: R

Figure 26-16 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 26-11 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
CC	[31:16]	The number of countable conditions present in the system

27 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.

27.1 Timers

The processor timers are connected to a system clock signal that operates even when the ARCv3-based processor is in some (but not all) of the processor sleep states. 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 24-3 on page 1031.

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 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.

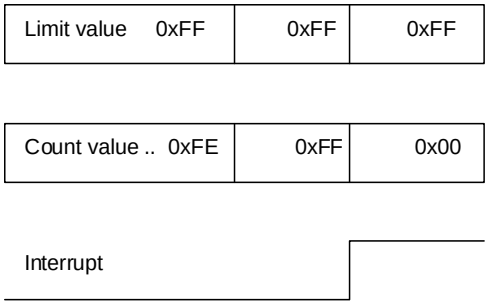
27.1.1 Programming

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.

Figure 27-1 Interrupt Generated after Timer Reaches Limit Value



The software must clear the timer interrupts by clearing CONTROLn.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.

27.2 Auxiliary Timer Registers

The ARCv3 architecture provides one or two optional cycle counters, each of which defines three additional auxiliary registers as listed in Table 27-1 and Table 27-2.

Table 27-1 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 27-2 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 27-3 Auxiliary Register for Real-time counter

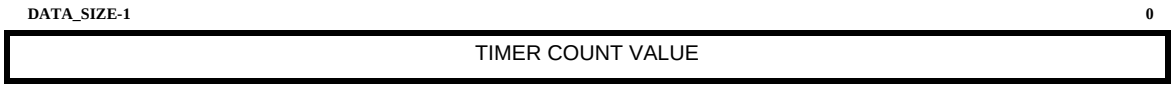
Address	Auxiliary Register Name	Description
0x103	RTC Control Register, AUX_RTC_CTRL	RTC control register
0x104	RTC Count Register, AUX_RTC_LOW	RTC count low register

27.2.1 Timer 0 Count Register, COUNT0

Address: 0x21

Access: RW

Figure 27-2 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.

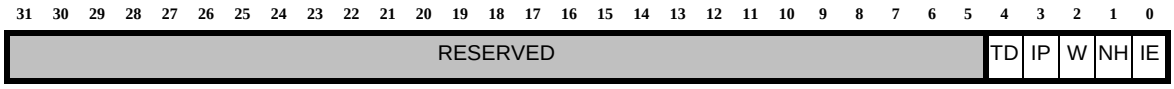
This register contains `0x00000000` on reset.

27.2.2 Timer 0 Control Register, CONTROL0

Address: 0x22

Access: RW

Figure 27-3 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 ARCv3-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 27-4 TIMER 0 Field Descriptions

Field	Bit	Description
IE	[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	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]	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 27-4 TIMER 0 Field Descriptions (Continued)

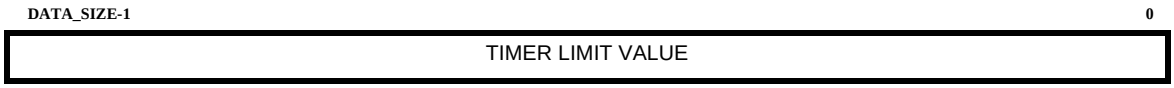
Field	Bit	Description
IP	[3]	The IP bit is set when the COUNTn register reaches the LIMITn value, and remains set until cleared by the timer interrupt service routine. The IP bit can be cleared by re-writing the desired values of W, NH and IE into the CONTROL0 register, thereby writing a 0 into the IP 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]	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.

27.2.3 Timer 0 Limit Register, LIMIT0

Address: 0x23

Access: RW

Figure 27-4 Timer 0 Limit Value Register



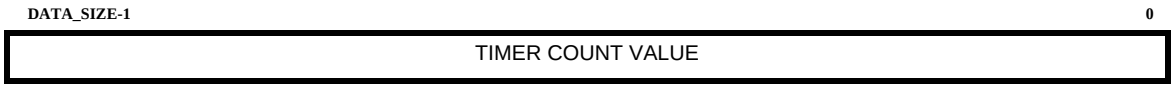
You must write the limit value into this register. The limit value is the value after which an interrupt or a reset must be generated. For backward compatibility to previous processor variants, the timer limit register is set to 0x00FFFFFF when the processor is Reset.

27.2.4 Timer 1 Count Register, COUNT1

Address: 0x100

Access: RW

Figure 27-5 Timer 1 Count Value Register



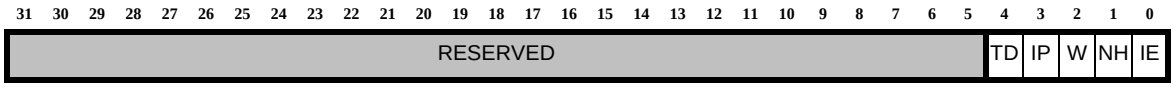
See Timer 0 Count Register, COUNT0 on page 1100 for field information.

27.2.5 Timer 1 Control Register, CONTROL1

Address: 0x101

Access: RW

Figure 27-6 Timer 1 Control Register



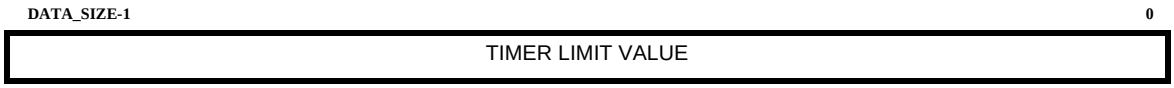
See Timer 0 Control Register, CONTROL0 on page 1101 for field information.

27.2.6 Timer 1 Limit Register, LIMIT1

Address: 0x102

Access: RW

Figure 27-7 Timer 1 Limit Value Register



See Timer 0 Limit Register, LIMIT0 on page 1103 for field information.

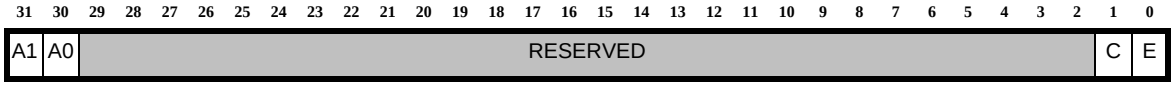
27.2.7 RTC Control Register, AUX_RTC_CTRL

Address: 0x103

Access: rW

Reset: 0x0000_0000

Figure 27-8 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 27-5 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. C is always read as zero.
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 Register, AUX_RTC_LOW registers. A0 and A1 are always read as zero and are ignored on write.

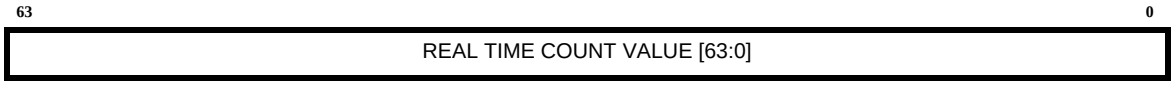
27.2.8 RTC Count Register, AUX_RTC_LOW

Address: 0x104

Access: r

Reset: 0x0000_0000_0000_0000

Figure 27-9 RTC Count Low Register



The RTC Count Low register is a 64-bit read-only register that returns the value of the entire RTC counter when read using an LRL instruction.

27.2.9 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 27-10 TIMER_BUILD Register

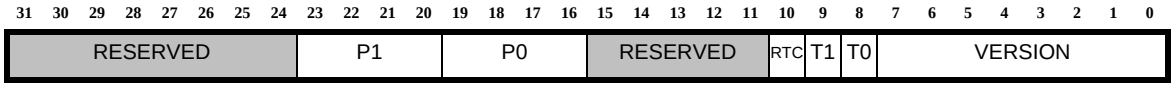


Table 27-6 TIMER_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	Current version. All non-enumerated values are reserved. ■ 0x7: 64-bit timers
T0	[8]	Timer 0 Present ■ 0x0: No timer 0 ■ 0x1: Timer 0 present
T1	[9]	Timer 1 Present ■ 0x0: No timer 1 ■ 0x1: Timer 1 present
RTC	[10]	64-bit RTC Configuration ■ 0x0: 64-bit RTC is disabled. ■ 0x1: 64-bit RTC is enabled.
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.

28 ARC Trace

28.1 ARC Trace Introduction

ARC Trace is a real-time state tracing system for ARCv3-based processors. ARC Trace allows you to trace the following state information for a processor:

- Data collection: data read and write to auxiliary registers, core register writes, and memory read and writes.
- Debug trace: Program flow, process ID trace, and watchpoint trace.

You can use the MetaWare debugger to enable ARC Trace and display the debug trace.

You can use the ARC Trace control registers to program the ARC Trace at set-up time. These registers are not common to all producers. Each producer has an independent set of control registers.

28.2 ARC Trace Auxiliary Registers

Table 28-1 ARC Trace Registers

Address	Auxiliary Register Name	Description
0xF2	ARC Trace Build Configuration Register, RTT_BUILD	Build configuration register
0x380	ARC Trace Address Register, RTT_ADDRESS	Contains the ARC Trace memory address that you want to access
0x381	ARC Trace DATA Register, RTT_DATA	Contains the data that you want to read or write to ARC Trace
0x382	ARC Trace CMD Register, RTT_COMMAND	ARC Trace command register

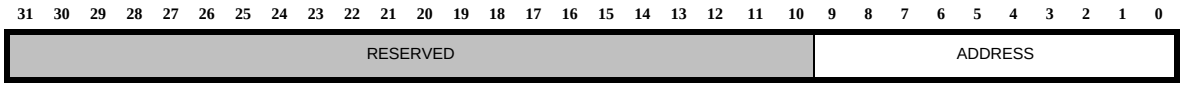
28.2.1 ARC Trace Address Register, RTT_ADDRESS

Address: 0x380

Access: RW

Reset: 0x0000_0000

Figure 28-1 RTT_ADDRESS Register



This register specifies the ARC Trace memory address which you want to write to or read from. The address written to this register is presented on the ARC Trace interface address bus; the desired address must be set up prior to writing the RTT_COMMAND register, which initiates the ARC Trace read or write operation.

The operation that the processor performs on this ARC Trace memory address is specified by the RTT_COMMAND register.

The RTT_DATA register contains the data read from this ARC Trace memory address or the data you want to write to this memory address.

For more information about this register, see the *ARC Trace Databook*.

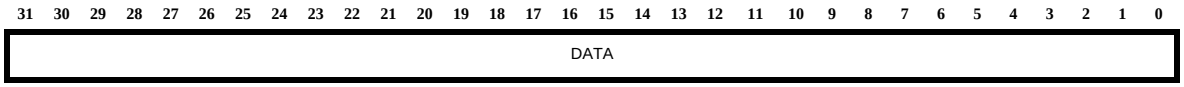
28.2.2 ARC Trace DATA Register, RTT_DATA

Address: 0x381

Access: RW

Reset: 0x0000_0000

Figure 28-2 RTT_DATA Register



This register contains the data read from the ARC Trace or the data to be written to the ARC Trace.

The read or write operation is determined by the value of the RTT_COMMAND register.

The RTT_ADDRESS register specifies the ARC Trace memory address that is being accessed.

For more information about this register, see the *ARC Trace Databook*.

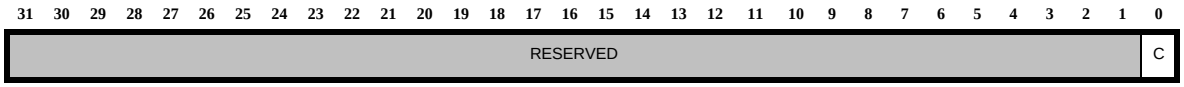
28.2.3 ARC Trace CMD Register, RTT_COMMAND

Address: 0x382

Access: r/w

Reset: 0x0000_0000

Figure 28-3 RTT_COMMAND Register



This register specifies the actions that the processor performs on ARC Trace.

Table 28-2 RTT_COMMAND Bit Field Description

Field	Bit	Description
C	[0]	**Specifies the action performed on ARC Trace** ■ 0: indicates a read operation from ARC Trace ■ 1: indicates a write operation to ARC Trace

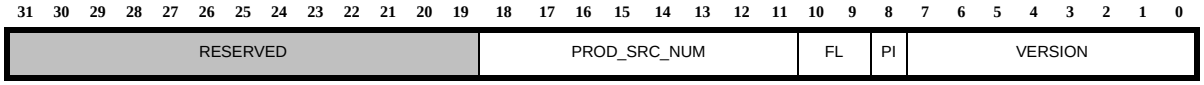
For more information about this register, see the *ARC Trace Databook*.

28.2.4 ARC Trace Build Configuration Register, RTT_BUILD

Address: 0xF2

Access: R

Figure 28-4 RTT_BUILD Register



This register specifies the build-time configuration of ARC Trace.

Table 28-3 RTT_BUILD Field Descriptions

Field	Bit	Description
VERSION	[7:0]	**Version of ARC Trace** ■ `0x1`: Initial version of ARC Trace ■ `0x2`: Second version of ARC Trace ■ `0x3`: Third version of ARC Trace ■ `0x4`: Fourth version: - Added support for CoreSight™ - Modified trace buffers in CoreSight builds to use flip flop arrays not RAMs. ■ `0x5`: Fifth version. Indicates support for both nexus interface and CoreSight. However, you can configure only one interface in a processor build. ■ `0x6`: Sixth version of ARC Trace ■ `0x7`: Seventh version of ARC Trace for ARCv3-based processors
PI	[8]	**Indicates the presence of ARC Trace programming interface** This bit is always set to 1; it means that you can program the ARC Trace control registers using this core.
FL	[10:9]	**Indicates the feature level of the producer** ■ `0x0`: small producer; defines program counter as a source for trace messages ■ `0x1`: medium producer; defines program counter and memory read/write as sources for trace messages ■ Other values are: reserved

Table 28-3 RTT_BUILD Field Descriptions (Continued)

Field	Bit	Description
PROD_SRC_NUM	[18:11]	**Indicates the number assigned to this core by ARC Trace in a multi-core configuration.** Valid values: 0, 1, 2, 3. Other values are reserved. When CoreSight™ is supported, this field is always set to 0.
RESERVED	[31:19]	Reserved. Read-as-zero; ignored on write.

For more information about this register, see the *ARC Trace Databook*.

28.3 ARC Trace Address Map

ARCv3 ISA has the following kinds of registers:

- ARC Trace Producer Registers
- ARC Trace Transport Status and Control Registers

ARC Trace Address Map is shown in Table 28-4.

Table 28-4 ARC Trace Address Map

Bank	Start Address	End Address
Producer registers	0x000	0x04F
Transport status and control registers	0x050	0x05F

28.4 ARC Trace Producer Registers

Each producer has instances of the following ARC Trace producer registers. The number of producer register banks is equal to the number of producers configured during build-time in ARChitect.

The Producer Registers allows the capturing probe JTAG interface to manipulate the producer configurations. Only one register can be programmed at a time and the Status Read only registers hold the status of the ARC Trace internal Modules.

Table 28-5 summarizes the build configuration registers, control registers, and status registers for components that are described in this document.

Table 28-5 ARC Trace Producer Registers

Offset from Base Address (0x00)	Name	Registers Present in the Configurations
0x00	ARC Trace I/F Build Configuration Register, ARCT_BUILD	All

Table 28-5 ARC Trace Producer Registers (Continued)

Offset from Base Address (0x00)	Name	Registers Present in the Configurations
0x01	Trace Producer Select Register, ARCT_PRSEL	All
0x02	Trace Nexus Flush Command Register, ARCT_FLUSH_COMMAND	All
0x04	Trace Timestamp Enable Register, ARCT_TSTAMP_ENABLE	All
0x05	Trace Debugger Message Enable Register, ARCT_DEBUGGER_MESSAGE_ENABLE	All
0x06	Producer System Timestamp Enable Register, PR_SYTM_ENABLE	With Nexus offload With on-chip memory
0x06	Producer CoreSight Timestamp Register, CSTEN_REG_ADDR	with CoreSight ATB
0x07	Producer Cross Trigger Enable Register, CTIEN_REG_ADDR	with CoreSight ATB
0x0A	Producer Trace Attribute Register, ARCT_PRATTR	All
0x0B	Trace Producer Source Enable Register, PR_SRC_EN	All Small: 2 bits Medium: 4 bits Full: 7 bits
0x0D	EVTI Control Register, PR_EVTI_REG	With Nexus offload
0x10	ARC Trace Producer Build Configuration Register, RTT_PRDCR_BCR	All
For 0x20 to 0x27 See Table 28-18	Address Filter Registers	All
0x30	Trace Producer Type Register, ARCT_PRTYPE	All
0x31	Trace Filter Type Register, ARCT_FLT_TYPE	All
0x32	Trace Filter Control Register, ARCT_FLT_CTRL	All
0x34	Trace Producer Watchpoint Status Register, PR_WP_STATUS	All
0x35	Trace Producer Watchpoint Enable Register, PR_WP_ENABLE	All

28.4.1 Register Description

ARC Trace supports programmable eight address filters and two data filters. Each address filter can be programed as a global filter, a range filter, or a trigger filter.

A range filter captures the values between the start address and the stop address including the start and stop addresses of a producer's source.

A global filter is a trace-all filter of PC range. Global filter captures the messages of all sources which fall within the global filter PC start and PC stop range.

A trigger filter captures the values from the matched start register address to stop register address including the interrupts and exceptions which are triggered in the range. The trigger register needs to be programed to select the operation of the filters between range filter and trigger filter.

To select the filter, each range/trigger filter requires a pair of registers to hold the start address and stop address of the source. ARC Trace has eight filter start address registers and eight filter stop registers named as Address Start Filter0 – Address Start Filter7 and Address Stop Filter0 – Address Stop Filter7. Multiple filters can be programmed to the same source. Assigning a range filter to a producer source is done by programming the filter type register.

When a global filter is programmed as a trigger filter, it starts capturing the trace from the start address of the global trigger filter and stops when it matches to stop address global trigger.

If both global trigger and a source trigger are active, source trigger is active only in global range. The source trigger is disabled when the address matches to the stop address of the source or it is out of the global filter range. The programming and its use cases are explained in Global filters section.

ARC Trace data filters named as Data Filter0 and Data Filter1 hold the data of the source to be matched. ARC Trace generates the Nexus message when trace data matches with the filter data. The data filters support both 32-bit and 64-bit data width. If data width is 64-bit, the data filters have two 32-bit registers to hold the 64-bit data that is, Data Filter(0/1) LSW and Data Filter(0/1) MSW, else only data filter(0/1) LSW exist.

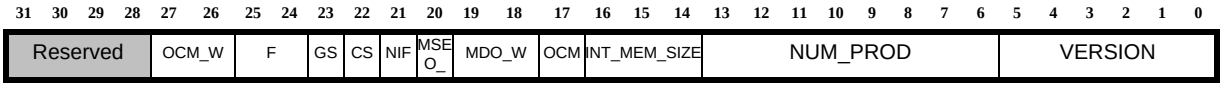
28.4.2 ARC Trace I/F Build Configuration Register, ARCT_BUILD

Address: 0x00

Access: R

Reset: Build-time configuration values from ARChitect

Figure 28-5 ARCT_BUILD Register



This register specifies the build-time configuration of ARC Trace. This information is presented as an aid to connected debuggers to quickly identify the trace resources and message content, format, stream interface.

Table 28-6 ARCT_BUILD Field Descriptions

Field	Bits	Description
VERSION	[7:0]	**Version of ARC Trace** ■ `0x07` = Support for ARCv3-based processors
NUM_PROD	[13:6]	**Number of trace producers. Max: 256**
INT_MEM_SIZE	[16:14]	**Internal trace memory size. All non-enumerated values are reserved.** ■ `0x0`:4K ■ `0x1`: 8K ■ `0x2`: 16K ■ `0x3`: 32K ■ `0x4`: 64K ■ `0x5`: 128K
OCM	[17]	**On-Chip Memory Support** ■ `0x1`: On-Chip Memory Present ■ `0x0`: On-Chip Memory Not Present For CoreSight builds. This bit always reads `0`.
MDO_W	[19:18]	**Number of Nexus ports present in the build. All non-enumerated values are reserved.** ■ `0x1`:Single Nexus port ■ `0x2`: Dual Nexus ports
MSEO_W	[20]	**Nexus MSEO Width** ■ `0x0`:1-bit ■ `0x1`: 2-bit

Table 28-6 ARCT_BUILD Field Descriptions (Continued)

Field	Bits	Description
NIF	[21]	**Nexus interface present** ■ 0x1: Nexus Interface present ■ 0x0: Nexus Interface not present For CoreSight builds. This bit always reads 0.
CS	[22]	**CoreSight compatible (ATB) interface present.** ■ 0x1: Coresight Interface present ■ 0x0: Coresight Interface not present
GS	[23]	**Graduating store feature present**. The data trace write message is affected if this feature is configured. ■ 0x1: Graduating Store present ■ 0x0: Graduating Store not present
F	[25:24]	**Address Filter Pairs** ■ 0x0: 8 ■ 0x1: 4 ■ 0x2: 2 ■ 0x3: 0
OCM_W	[27:26]	**On-Chip Memory width** ■ 0x0:32 bit ■ 0x1: 64 bit ■ 0x2: 128 bit For CoreSight builds. this bit always reads 00.
RESERVED	[31:28]	Reserved. Read-as-zero; ignored on write.

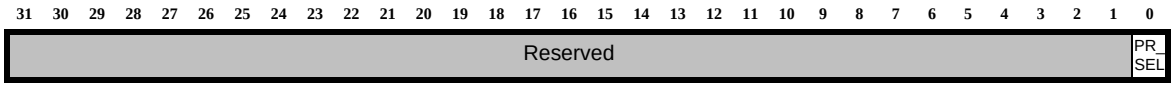
28.4.3 Trace Producer Select Register, ARCT_PRSEL

Address: 0x01

Access: RW

Reset: 0x0

Figure 28-6 ARCT_PRSEL Register



Enables trace collection for this producer.

Table 28-7 ARCT_PRSEL Field Descriptions

Field	Bits	Reset	Description
PR_SEL	[0]	0	**Producer Select** ■ `0x1`: Producer is active ■ `0x0`: Producer is not active
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

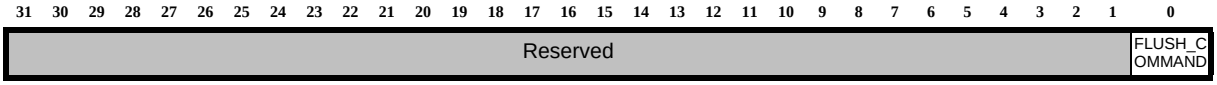
28.4.4 Trace Nexus Flush Command Register, ARCT_FLUSH_COMMAND

Address: 0x02

Access: RW

Reset: 0x0

Figure 28-7 ARCT_FLUSH_COMMAND Register



This register is used to flush internal buffers and places enqueued messages onto the trace offload interface. Initiate the flush operation by writing a 1 to FLUSH_COMMAND. This bit is automatically cleared to 0 when all stored messages have egressed and forwarded to Trace Transport Unit.

Note Completion of this operation does not mean generated that the messages are forwarded to the egress interface. For completion of flush operation, you must read the Flush Command Register of the Trace Transport Unit.

Table 28-8 ARCT_FLUSH_COMMAND Field Descriptions

Field	Bits	Reset	Description
FLUSH_COMMAND	[0]	0	Flush command bit for Nexus Aux/OCM offload. ■ `0x0`: Flush command completed. Writing 0 has no effect. ■ `0x1`: Assert 1 to initiate Flush operation. ■ Transition from 0x1 to 0x0 notifies the Trace client that the flush operation is completed.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

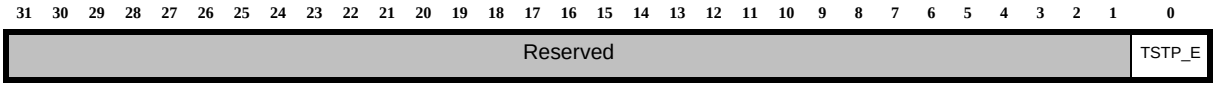
28.4.5 Trace Timestamp Enable Register, ARCT_TSTAMP_ENABLE

Address: 0x04

Access: RW

Reset: 0x0

Figure 28-8 ARCT_TSTAMP_ENABLE Register



Trace messages from this producer may include the optional nexus timestamp field with each message.

Table 28-9 ARCT_TSTAMP_ENABLE Field Descriptions

Field	Bits	Reset	Description
TSTP_E	[0]	0	Producer Timestamp Enable ■ `0x0`: Timestamp field is not included in trace messages. ■ `0x1`: Timestamp is included in trace messages.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

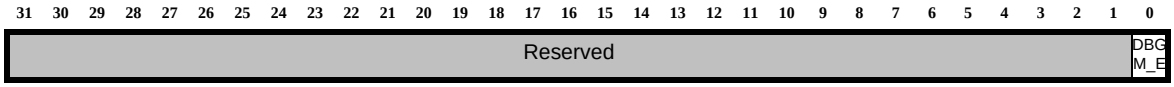
28.4.6 Trace Debugger Message Enable Register, ARCT_DEBUGGER_MESSAGE_ENABLE

Address: 0x05

Access: RW

Reset: 0x0

Figure 28-9 ARCT_DEBUGGER_MESSAGE_ENABLE Register



Debug Message enable. Enables you to trace the debugger-inserted instructions.

Table 28-10 ARCT_DEBUGGER_MESSAGE_ENABLE Field Descriptions

Field	Bits	Reset	Description
DBGM_E	[0]	0	Enable ARC Trace to trace debugger-inserted instructions ■ `0x0`: Debugger transactions are not included in trace stream for this producer. ■ `0x1`: Debugger transactions are included in trace stream for this producer.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

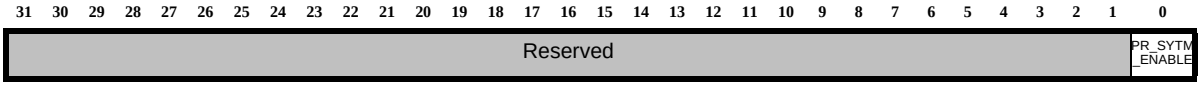
28.4.7 Producer System Timestamp Enable Register, PR_SYTM_ENABLE

Access: RW

Address: 0x06

Reset: 0

Figure 28-10 Producer CoreSight Timestamp Register, PR_SYTM_ENABLE



Set bit 0 to 1 to generate the system timestamp message when a qualifying condition is met. This register exists in builds with the on-chip memory or Nexus interface.

Table 28-11 Producer Cross Trigger Enable Register PR_SYTM_ENABLE Description

Field	Bits	Reset	Description
PR_SYTM_ENABLE	[0]	0	Set bit 0 to 1 to enable the ATB CoreSight Timestamp message
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

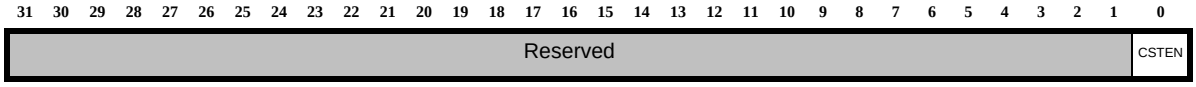
28.4.8 Producer CoreSight Timestamp Register, CSTEN_REG_ADDR

Access: RW

Address: 0x06

Reset: 0

Figure 28-11 Producer CoreSight Timestamp Register CSTEN_REG_ADDR



Set bit 0 to 1 to enable the ATB CoreSight Timestamp message. This register exists in builds with the CoreSight offload option.

Table 28-12 Producer Cross Trigger Enable Register CSTSEN_REG_ADDR Description

Field	Bits	Reset	Description
CSTEN	[0]	0	Set bit 0 to 1 to enable the ATB CoreSight Timestamp message
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

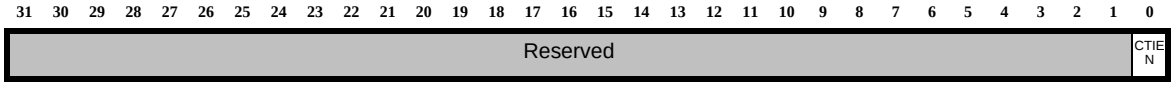
28.4.9 Producer Cross Trigger Enable Register, CTIEN_REG_ADDR

Access: RW

Address: 0x07

Reset: 0

Figure 28-12 Producer Cross Trigger Enable Register, CTIEN_REG_ADDR



Set bit 0 to enable the external cross trigger interface trace start and stop inputs to control tracing. When enabled, these inputs override any global filters that have been set. This register exists in builds with the CoreSight offload option.

Table 28-13 Producer Cross Trigger Enable Register Description

Field	Bits	Reset	Description
CTIEN	[0]	0	Set bit 0 to 1 to enable the external trace start and stop control.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

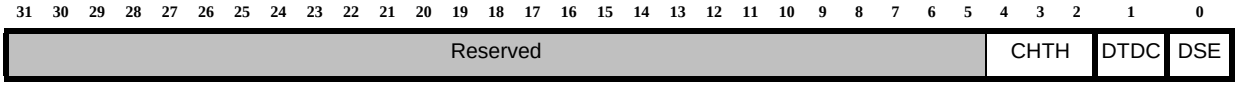
28.4.10 Producer Trace Attribute Register, ARCT_PRATTR

Access: RW

Address: 0x0A

Reset: 0x0

Figure 28-13 Producer Trace Attribute Register



Use this register to control the producer trace attributes such as: data suppression, including cache miss or hit information in a data trace message, and thresholds for generating a Resource Full message for conditional branches.

Table 28-14 Producer Trace Attribute Register Description

Field	Bits	Reset	Description
DSE	[0]	0	This bit is present only when ATB offload is configured. Control for data suppression feature, prioritizing program trace. Set to 1 to enable, 0 to disable. Use this bit to enable the data suppression feature. When this bit is set to 1 and both program and data tracing are enabled, data tracing is suppressed when the ATB FIFO reaches a defined threshold to try to prevent an overflow and subsequent loss of all trace information while preserving the program trace features.
DTDC	[1]	0	■ Data Trace Data Configuration. This bit is available only for medium producers. ■ 0x1: The DATA field in the data trace message is a 1-bit field that indicates a cache miss (1) or cache hit (0). For a graduated store, the DATA field is an 8-bit field with the cache miss or hit information in DATA[8] and the tag in DATA[7:0]. ■ 0x0: The DATA field in the data trace message indicates the data written to the memory or the graduation tag if the store graduates.
CHTH	[4:2]	0	Conditional History Threshold. Resource Full Messages issued for Conditional History overflow can be throttled, from the default setting, to allow finer grain instruction cycle calculations from the collected trace stream. For single-issue cores: ■ `0x2`: RFM is generated for every 2nd branch ■ `0x3`: RFM is generated for every 4th branch ■ `0x4`: RFM is generated for every 8th branch ■ `0x5`: RFM is generated for every 16th branch ■ `0x6`: RFM is generated for every 16th branch ■ `0x7`: RFM is generated for every 16th branch All other values: RFM is generated every 29th branch

Table 28-14 Producer Trace Attribute Register Description (Continued)

Field	Bits	Reset	Description
RESERVED	[31:5]	0	Reserved. Read-as-zero; ignored on write.

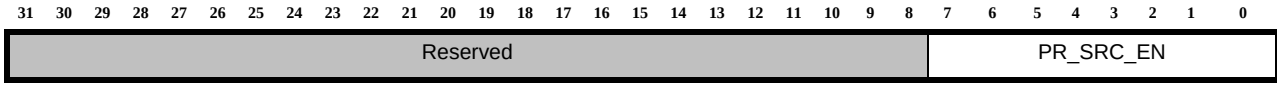
28.4.11 Trace Producer Source Enable Register, PR_SRC_EN

Access: RW

Address from 0x40: 0x0B

Reset: 0x01

Figure 28-14 Producer Source Enable Register



Producer Source Enable register holds the active sources of the producer.

Table 28-15 Producer Source Enable Description

Field	Bits	Reset	Description
PR_SRC_EN	[7:0]	1	Enable a source in a producer ■ bit [0]: set this bit to enable program counter as a source. ■ bit [1]: set this bit to enable memory write data as a source. ■ bit [2]: set this bit to enable memory read data as a source. ■ bit[6]: set this bit to enable debug status messages. When this bit is enabled, ARC Trace generates the Debug Status Trace. Write 0 to this bit to disable debug status messages. This bit is available in all producer configurations. All other values are reserved.
RESERVED	[31:8]	0	Reserved. Read-as-zero; ignored on write.

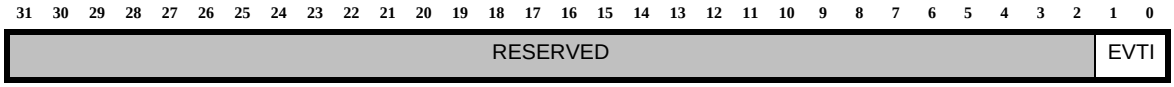
28.4.12 EVTI Control Register, PR_EVTI_REG

Address: 0x0D

Access: RW

Reset: 0x00

Figure 28-15 PR_EVTI_REG Register



This register lets you control ARC Trace behavior when an EVTI signal is asserted from the off-chip Nexus interface. This register is present only if ARC Trace is built with a Nexus interface. This register is reset when Trace Client sends synchronization message.

Note

- Writing 0 to this register is ignored
- Read to this register always returns 0.
- External Debugger and Software can not access this register.

Table 28-16 PR_EVTI_REG Field Descriptions

Field	Bits	Reset	Description
EVTI	[1:0]	0	**Enable or disable synchronization messages when an EVTI signal is asserted.** ■ `0x0`: Synchronize trace messages. The first message of each message type after the EVTI signal assertion is a sync message, For example, if ARC Trace is set to send sync messages on an EVTI signal, the first message after the EVTI assertion of the following trace messages will be a sync message: program trace, data trace read/write, program trace indirect branch history, auxiliary register read/write trace ■ `0x2`: Disabled. The EVTI signal is ignored. Other values of this field are reserved.
RESERVED	[31:1]	0	**Reserved. Read as zero. Ignore on writes**

28.4.13 ARC Trace Producer Build Configuration Register, RTT_PRDCR_BCR

Address: 0x10

Access: R

Figure 28-16 RTT_PRDCR_BCR Register

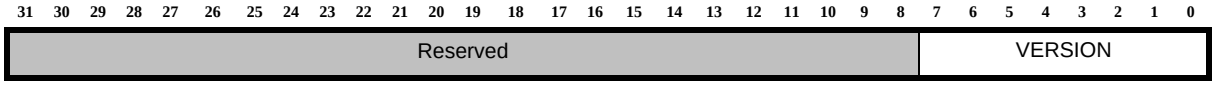


Table 28-17 RTT_PRDCR_BCR Description

Field	Bits	Description
VERSION	[7:0]	Version of Trace Producer ■ `0x0` = No additional feature is implemented
RESERVED	[31:8]	Reserved. Read-as-zero; ignored on write.

28.4.14 Address Filter Registers

ARC Trace supports up to eight address filters. The number of filers included in the build, is configurable at built time. Use the `-rtt_num_filters` option to configure the number of filters.

- The address filters can be programmed to capture the data of the following sources:Program counter
- Memory Write Data
- Memory Read Data

Data filters support all the sources except Program Counter.

When the filter registers are programmed for a source, the programming filters called with the source name as listed below:

- Program counter Filter start register
- Program counter Filter stop register
- Memory Write Filter start register
- Memory Write Filter stop register
- Memory Read Filter start register
- Memory Read Filter stop register

Each address filter has a start and stop address register associated with it. Table 28-18 shows the start address and stop address filters.

Table 28-18 Start Address Register and Stop Register Address Offsets

Filter	Start Address Register Offset	Stop Address Register Offset
Filter0	0x020	0x021
Filter1	0x022	0x023
Filter2	0x024	0x025
Filter3	0x026	0x027

28.4.14.1 Start Address Register, ARCT_ADSTR_FLTx

Access: RW

Reset: 0x00

Figure 28-17 Start Address Register

The PC start address register holds the value of the starting address of the PC for the range filter. A Range filter is a filter captures the values between the start value and stop value of a producer's source including the boundaries. The Range filter starts capturing the trace soon after the PC address value is matched or greater than the PC filter start address. The address filter start address must always be less than or equal to the stop address.

Note If PC_SIZE is less than 32 bits, when programming this register, set the MSB [32:PC] bits to 0.

Table 28-19 Start Address Register Field Descriptions

Field	Bits	Reset	Description
START	[ADDR_SIZE-1:0]	0	Filter's start address value

28.4.14.2 Trace Address Stop Filter Register, ARCT_ADSTP_FLTx

Access: RW

Reset: 0xFFFF_FFFF

Figure 28-18 Trace Address Stop Filter Register

This register holds the stop address for an address filter (range and trigger). The Range filter stops capturing the trace soon after the PC address value is matched or greater than the PC filter stop address.

Note

- Ensure that the stop address of a filter for program trace does not point to a delay-slot instruction.
- If PC_SIZE is less than 32 bits, when programming this register, set the MSB [32:PC] bits to 0.

Table 28-20 Stop Address Register Field Descriptions

Field	Bits	Reset	Description
ADDR	[ADDR_SIZE-1:0]	0xFFFF_FFFF	Filter's stop address value

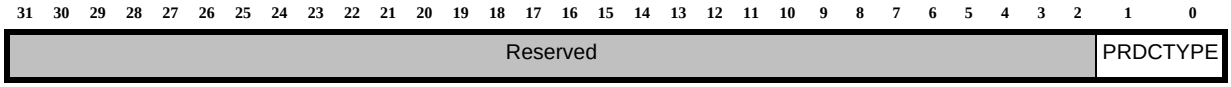
28.4.15 Trace Producer Type Register, ARCT_PRTYPE

Access: R

Address: 0x30

Reset: Inherited from the ARC HS6x build-time configuration (RTT Feature Level)

Figure 28-19 Producer Type Register



Holds information about the of the Producer ID and the type of producer messages that are generated.

Table 28-21 Producer Type Register Field Descriptions

Field	Bits	Reset	Description
PRDCTYPE	[1:0]	0	**Producer Type Register. All non-enumerated values are reserved.** ■ `0x0`: Small producer. Trace messages are suitable for program flow reconstruction. ■ `0x1`: Medium producer. Includes Trace messages of Small Producer type and Data memory transactions. ■ `0x2`: Full Producer.
RESERVED	[31:2]	0	Reserved. Read-as-zero; ignored on write.

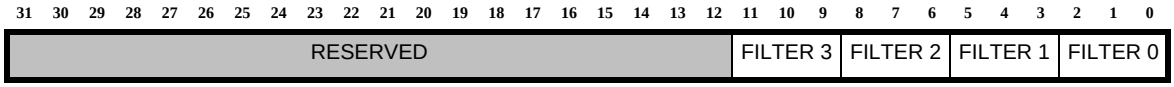
28.4.16 Trace Filter Type Register, ARCT_FLT_TYPE

Access: RW

Offset: 0x31

Reset: 0x00; All filters are disabled

Figure 28-20 Filter Source Select Register



Filter type register selects the filters for the producer sources. Set of three bits of filter type register represents a range filter of a source. Multiple filters can be assigned to a single source by writing the address filter filed source selection value in the filter programming field.

Data filters enabled by writing the data filter filed source selection value in the filter programming field. For more information about filters, see "Address Filter Registers" on page 1132.

For memory operations, when the MSB (bit [27] for Data Filter0 and bit [31] for Data Filter1) is set to 1'b1, the data is compared on both read and write paths.

For core writes, when MSB (bit [27] for Data Filter0 and bit [31] for Data Filter1) is set to 1'b1, 64-bit data comparisons are enabled, otherwise 32-bit data comparisons are enabled.

Table 28-22 Filter Source Select Register Description

Field	Reset	Description
FILTER [n]	0	**To assign a filter to a producer's source, the respective bits should be programmed as shown below. All non-enumerated values are reserved.** ■ `0x0` Filter is disabled ■ `0x1` Program counter filter ■ `0x2` Memory write filter (medium producer) ■ `0x3` Memory read filter (medium producer) ■ `0x7:` Global filter

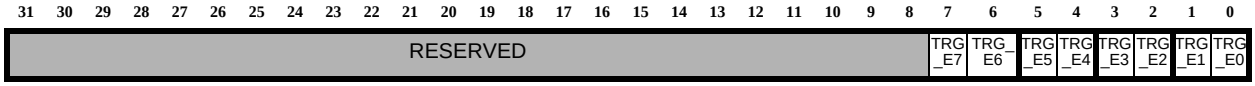
28.4.17 Trace Filter Control Register, ARCT_FLT_CTRL

Access: RW

Offset: 0x32

Reset: 0x00

Figure 28-21 Trace Filter Control Register



Selects the filter as a range filter or a trigger filter.

Table 28-23 Trace Filter Control Register

Field	Bits	Reset	Description
TRG_E[0]	[0]	0	Producer filter 0 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[1]	[1]	0	Producer filter 1 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[2]	[2]	0	Producer filter 2 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[3]	[3]	0	Producer filter 3 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[4]	[4]	0	Producer filter 4 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[5]	[5]	0	Producer filter 5 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[6]	[6]	0	Producer filter 6 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.
TRG_E[7]	[7]	0	Producer filter 7 trigger enable 1: Filter used as trigger filter. 0: Filter used as range filter.

Table 28-23 Trace Filter Control Register

Field	Bits	Reset	Description
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

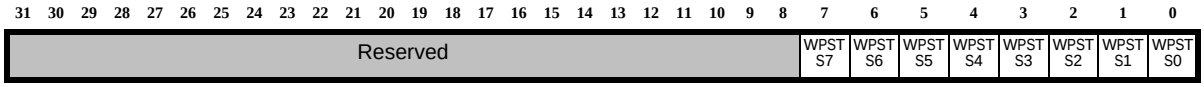
28.4.18 Trace Producer Watchpoint Status Register, PR_WP_STATUS

Access: R

Address: 0x34

Reset: 0x00

Figure 28-22 Producer Watchpoint Status Register



Each producer supports up to 8 watch points. The Producer actionpoints are mapped to the producer Watchpoint status register. The watch point messages are generated based on the bit set from the Watchpoint source of that producer.

Table 28-24 Producer Source Watchpoint Status Register Description

Field	Bits	Reset	Description
WPSTS0	[0]	0	Watchpoint Status 0 ■ 0x0: Producer actionpoint 0 de-asserted in this cycle ■ 0x1: Producer actionpoint 0 asserted in this cycle
WPSTS1	[1]	0	Watchpoint Status 1 ■ 0x0: Producer actionpoint 1 de-asserted in this cycle ■ 0x1: Producer actionpoint 1 asserted in this cycle
WPSTS2	[2]	0	Watchpoint Status 2 ■ 0x0: Producer actionpoint 2 de-asserted in this cycle ■ 0x1: Producer actionpoint 2 asserted in this cycle
WPSTS3	[3]	0	Watchpoint Status 3 ■ 0x0: Producer actionpoint 3 de-asserted in this cycle ■ 0x1: Producer actionpoint 4 asserted in this cycle
WPSTS4	[4]	0	Watchpoint Status 4 ■ 0x0: Producer actionpoint 4 de-asserted in this cycle ■ 0x1: Producer actionpoint 4 asserted in this cycle
WPSTS5	[5]	0	Watchpoint Status 5 ■ 0x0: Producer actionpoint 5 de-asserted in this cycle ■ 0x1: Producer actionpoint 5 asserted in this cycle
WPSTS6	[6]	0	Watchpoint Status 6 ■ 0x0: Producer actionpoint 6 de-asserted in this cycle ■ 0x1: Producer actionpoint 6 asserted in this cycle

Table 28-24 Producer Source Watchpoint Status Register Description (Continued)

Field	Bits	Reset	Description
WPSTS7	[7]	0	Watchpoint Status 7 ■ 0x0: Producer actionpoint 7 de-asserted in this cycle ■ 0x1: Producer actionpoint 6 asserted in this cycle
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

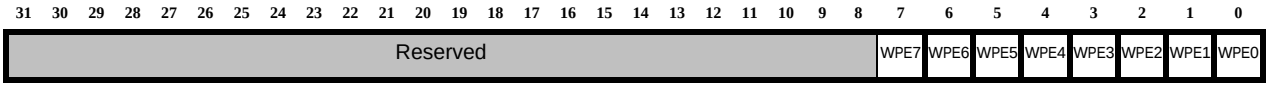
28.4.19 Trace Producer Watchpoint Enable Register, PR_WP_ENABLE

Access: RW

Address: 0x35

Reset: 0x000000FF

Figure 28-23 Producer Watchpoint Enable Register



This register holds the enable for watchpoint messages when corresponding producer actionpoint is set in Producer Watchpoint status register.

Table 28-25 Producer Watchpoint Enable Register Description

Field	Bits	Reset	Description
WPE0	[0]	1	Enable Watchpoint 0 trace ■ 0x0: Watchpoint trace 0 is disabled this cycle ■ 0x1: Watchpoint trace 0 is enabled this cycle
WPE1	[1]	1	Enable Watchpoint 1 trace ■ 0x0: Watchpoint trace 1 is disabled this cycle ■ 0x1: Watchpoint trace 1 is enabled this cycle
WPE2	[2]	1	Enable Watchpoint 2 trace ■ 0x0: Watchpoint trace 2 is disabled this cycle ■ 0x1: Watchpoint trace 2 is enabled this cycle
WPE3	[3]	1	Enable Watchpoint 3 trace ■ 0x0: Watchpoint trace 3 is disabled this cycle ■ 0x1: Watchpoint trace 3 is enabled this cycle
WPE4	[4]	1	Enable Watchpoint 4 trace ■ 0x0: Watchpoint trace 4 is disabled this cycle ■ 0x1: Watchpoint trace 4 is enabled this cycle
WPE5	[5]	1	Enable Watchpoint 5 trace ■ 0x0: Watchpoint trace 5 is disabled this cycle ■ 0x1: Watchpoint trace 5 is enabled this cycle
WPE6	[6]	1	Enable Watchpoint 6 trace ■ 0x0: Watchpoint trace 6 is disabled this cycle ■ 0x1: Watchpoint trace 6 is enabled this cycle

Table 28-25 Producer Watchpoint Enable Register Description (Continued)

Field	Bits	Reset	Description
WPE7	[7]	1	Enable Watchpoint 7 trace ■ `0x0`: Watchpoint trace 7 is disabled this cycle ■ `0x1`: Watchpoint trace 7 is enabled this cycle
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

28.5 ARC Trace Transport Status and Control Registers

The Trace Transport Status and Control registers allow the capture probe to manipulate ARC Trace. Only one register is programmed at a time and the Status Read only registers holds the status of the ARC Trace.

Table 28-26 ARC Trace Control Registers

Offset from Base Address (0x50)	Name	Registers Present in the Configurations
0x52	Trace Nexus Flush Command Register, ARCT_FLUSH_COMMAND	All
0x53	Producer CoreSight ID Register, ATID_REG_ADDR	With CoreSight ATB
0x54	Producer ATB SYNC Frame Insertion Register, SYNCRFR_REG_ADDR	With CoreSight ATB
0x54	Trace On-Chip Memory Base Address Register, ARCT_OCM_BASE	With on-chip memory
0x55	Trace On-Chip Memory Size Register, ARCT_OCM_SIZE	
0x56	Trace On-Chip Memory Write Pointer Register, ARCT_OCM_WPTR	
0x57	Trace Transport Status Register, ARCT_TR_STATUS	With on-chip memory or Nexus interface
0x58	Trace Nexus Offload Control Register, ARCT_OFFLOAD_CTRL	
0x59	Pattern Generation Register, PTRN_GEN	With Nexus offload
0x5A	Nexus Clock Control Register, NEXUS_CLK_DIV	

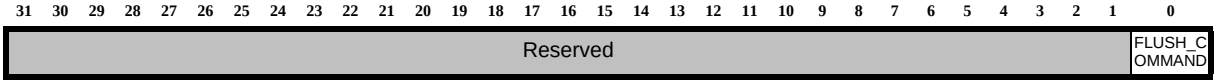
28.5.1 Trace Nexus Flush Command Register, ARCT_FLUSH_COMMAND

Address: 0x52

Access: RW

Reset: 0x0

Figure 28-24 ARCT_FLUSH_COMMAND Register



This register is used to flush internal buffers and places enqueued messages onto the trace offload interface. Initiate the flush operation by writing a 1 to FLUSH_COMMAND. This bit is automatically cleared to 0 when all stored messages have egressed and forwarded to Trace Transport Unit.

Note Completion of this operation does not mean generated that the messages are forwarded to the egress interface. For completion of flush operation, you must read the Flush Command Register of the Trace Transport Unit.

Table 28-27 ARCT_FLUSH_COMMAND Field Descriptions

Field	Bits	Reset	Description
FLUSH_COMMAND	[0]	0	Flush command bit for Nexus Aux/OCM offload. ■ `0x0`: Flush command completed. Writing 0 has no effect. ■ `0x1`: Assert 1 to initiate Flush operation. ■ Transition from 0x1 to 0x0 notifies the Trace client that the flush operation is completed.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

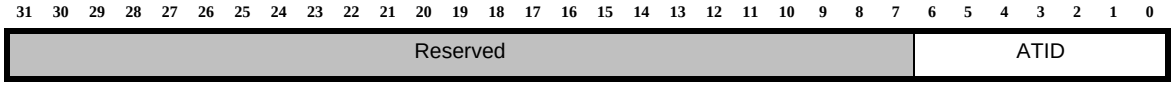
28.5.2 Producer CoreSight ID Register, ATID_REG_ADDR

Access: RW

Address: 0x53

Reset: 0x0

Figure 28-25 Producer CoreSight ID Register



Sets the ID on the CoreSight infrastructure for this producer. Each trace source in the system must be set to have a unique ID value.

Table 28-28 Producer CoreSight ID Register Description

Field	Bits	Reset	Description
ATID	[6:0]	1	[6:0] ID field output into ATB infrastructure for this producer
RESERVED	[31:7]	0	Reserved. Read-as-zero; ignored on write.

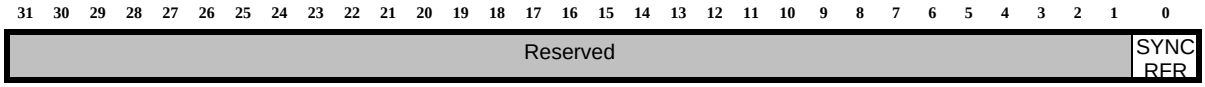
28.5.3 Producer ATB SYNC Frame Insertion Register, SYNCRFR_REG_ADDR

Access: RW

Address: 0x54

Reset: 0x0000

Figure 28-26 Producer ATB SYNC Frame Insertion Register SYNCREQ_REG_ADDR



This register can be used to insert SYNC frames (0xFF) into the CoreSight trace stream sent by the ARC Trace module into the CoreSight ATB infrastructure. These frames are used by the debugger to align the captured byte stream and allow reconstruction of the Nexus trace stream before processing. This register exists in builds with the CoreSight offload option. Only the Trace Transport Unit can access this register.

Note

- Writing 0 to this register is ignored.
- Read to this register always returns 0.
- External debugger and software can not access this register.

Table 28-29 Producer SYNC Frame Insertion Register Description

Field	Bits	Reset	Description
SYNCRFR	[0]	0	Enable or disable SYNC frame insertion in ATB stream.
RESERVED	[31:1]	0	Reserved. Read-as-zero; ignored on write.

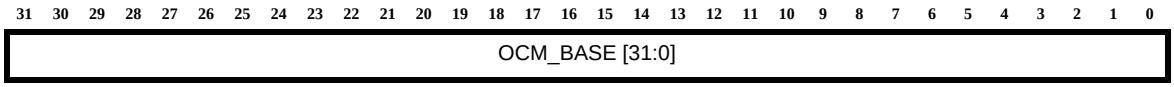
28.5.4 Trace On-Chip Memory Base Address Register, ARCT_OCM_BASE

Address: 0x54

Access: RW

Reset: 0x0

Figure 28-27 ARCT_OCM_BASE Register



ARCT_OCM_BASE holds the absolute address value for the on-chip memory where the AHB/AXI master stores the Nexus messages from trace. The address must be a 64-byte aligned address for 32-bit data busses and 128-byte aligned for 64-bit data busses.

Note If ADDR_SIZE is less than 32 bits, when programming this register, set the MSB [32:ADDR_SIZE] bits to `0`.

Table 28-30 ARCT_OCM_BASE Field Descriptions

Field	Bits	Description
OCM_BASE	[31:0]	**Base Address**

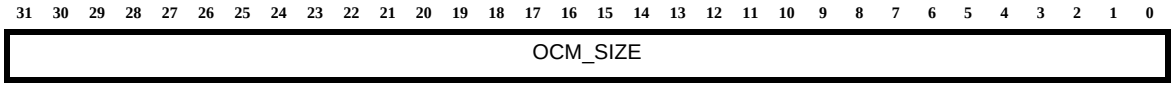
28.5.5 Trace On-Chip Memory Size Register, ARCT_OCM_SIZE

Address: 0x55

Access: RW

Reset: 0x0

Figure 28-28 ARCT_OCM_SIZE Register



ARCT_OCM_SIZE holds the size of the maximum addressable memory reserved for Trace On-Chip memory where Nexus messages are stored. The value of OCM_SIZE must be in multiples of ARCT_BUILD[Int_Mem_Size].

Table 28-31 ARCT_OCM_SIZE Field Descriptions

Field	Bits	Reset	Description
OCM_SIZE	[31:0]	0x8000 0000	**Memory size**

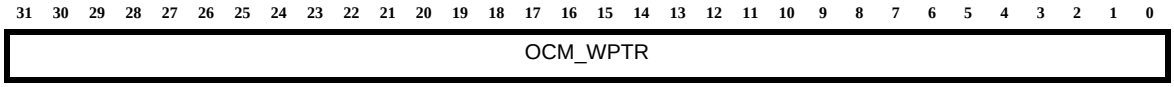
28.5.6 Trace On-Chip Memory Write Pointer Register, ARCT_OCM_WPTR

Address: 0x56

Access: RW

Reset: 0x0

Figure 28-29 ARCT_OCM_WPTR Register



ARCT_OCM_WPTR holds the write pointer value for On Chip Memory allocated for Trace. This register is updated when a write is issued to OCM. This register holds the relative address and the absolute location of OCM is calculated by adding this write-pointer value to OCM Base Address Register value.

The write pointer is a byte counter and its resolution is dependent on the memory-bus width of the producer. If the producer uses a 64-bit memory bus width, the write pointer is an eight-byte counter. If the producer uses a 32-bit memory bus width, the write pointer is a four-byte counter. When trace messages end in a non-aligned 8-byte location, ARC Trace appends zeros to the MSB bytes.

When wrap is disabled and OCM is full (`TR_STATUS[6]==1`), ARC Trace is halted and it cannot write to the on-chip memory. To unhalt ARC Trace, the write pointer is reset to point to the on chip memory base address. Before unhalting ARC Trace, ensure that you have read the on-chip memory to avoid trace data loss.

When wrap is enabled and wrapping occurs, trace data is overwritten.

Note If ADDR_SIZE is less than 32 bits, when programming this register, set the MSB [32:ADDR_SIZE] bits to `0`.

Table 28-32 ARCT_OCM_WPTR Field Descriptions

Field	Bits	Reset	Description
OCM_WPTR	[31:0]	0x0000 0000	**Current relative address of OCM write pointer.**

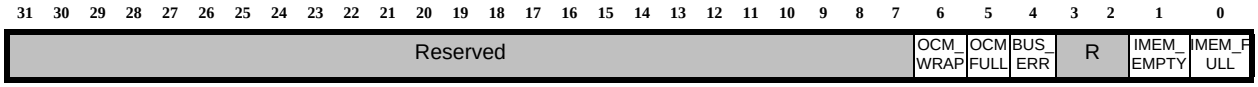
28.5.7 Trace Transport Status Register, ARCT_TR_STATUS

Address: 0x57

Access: R

Reset: 0x02

Figure 28-30 ARCT_TR_STATUS Register



Status of the Trace intermediate buffers and On-Chip Memory storage.

Table 28-33 ARCT_TR_STATUS Field Descriptions

Field	Bits	Reset	Description
IMEM_FULL	[0]	0	Internal Memory Full. ■ `0x1`: Full ■ `0x0`: Not full This field is valid for both on-chip memory offload and nexus offload configurations.
IMEM_EMPTY	[1]	0	Internal Memory Empty ■ `0x1`: Empty ■ `0x0`: Not empty This field is valid for both on-chip memory offload and nexus offload configurations.
RESERVED	[3:2]	0	Reserved. Read-as-zero; ignored on write.
BUS_ERR	[4]	0	Internal Memory Bus Error Detected. ■ `0x1`: Bus error detected. Bit is cleared when host reads Transport status register ■ `0x0`: No Bus error detected. This bit is cleared when the host (debugger) reads the ARCT_TR_STATUS register. This bit is valid for an ARC Trace configuration that includes an on-chip memory offload interface. For other ARC Trace configurations without the on-chip memory interface, this bit is reserved.
OCM_FULL	[5]	0	On-Chip Memory Full. ■ `0x1`: Full. ■ `0x0`: Not full

Table 28-33 ARCT_TR_STATUS Field Descriptions (Continued)

Field	Bits	Reset	Description
OCM_WRAP	[6]	0	On-chip memory wrap status ■ `0x1`: Write pointer has wrapped. ARC Trace inserts '1' in the MSB to each nexus message to make them word-aligned. ■ `0x0`: Write pointer has not wrapped. The nexus messages in this case are not word aligned. ARC Trace pads '0' in the MSB after the last nexus message.
RESERVED	[31:7]	0	Reserved. Read-as-zero; ignored on write.

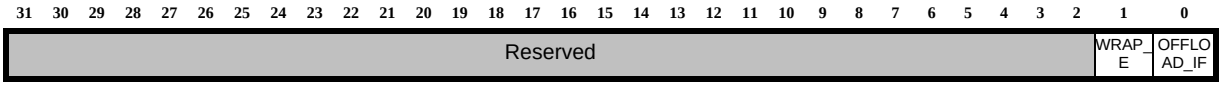
28.5.8 Trace Nexus Offload Control Register, ARCT_OFFLOAD_CTRL

Address: 0x58

Access: RW

Reset: 0x01

Figure 28-31 ARCT_OFFLOAD_CTRL Register



Selects the mode of storage for Nexus trace messages. The Nexus I/F and/or On-Chip Memory must be included in the build to be applicable. You can also enable the wrap feature for On-Chip memory. The wrap feature is a rolling storage buffer that replaces old data with incoming data. If wrap mode is not enabled, an internal memory full indicator is set and new data is lost when the On-Chip Memory is full.

Table 28-34 ARCT_OFFLOAD_CTRL Field Descriptions

Field	Bits	Reset	Description
OFFLOAD_IF	[0]	1	**Nexus interface Selection.** ■ `0x0`: Nexus interface selected for trace egress ■ `0x1`: On-chip memory interface is selected for trace storage
WRAP_E	[1]	0	**OCM Wrap mode enable.** ■ `0x0`: Internal memory wrap mode is disabled. ARC Trace stops offloading messages to the on chip memory until the debugger resets the write pointer to the OCM base address ■ `0x1`: Internal memory wrap mode is enabled.
Reserved	[31:2]	0	**Reserved. Read-as-zero; ignored on write.**

When the on-chip memory supports wrapping, the message formats are as follows:

```
128bit : {NTA_ID[3:0], MSEO[1:0], MDO[7:0], MSEO[1:0], MDO[7:0], MSEO[1:0], MDO[7:0],
            MSEO[1:0],  MDO[7:0],MSEO[1:0],  MDO[7:0],MSEO[1:0],  MDO[7:0],
          NTA_ID[3:0], MSEO[1:0], MDO[7:0], MSEO[1:0], MDO[7:0], MSEO[1:0], MDO[7:0],
            MSEO[1:0],  MDO[7:0],MSEO[1:0],  MDO[7:0],MSEO[1:0],  MDO[7:0]}
```

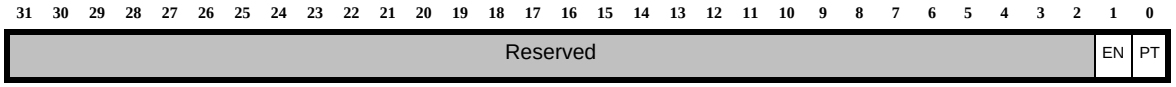
28.5.9 Pattern Generation Register, PTRN_GEN

Address: 0x59

Access: RW

Reset: 0x0

Figure 28-32 PTRN_GEN Register



This register lets you generate a bit pattern on the offload Nexus interface. You can use this bit pattern to tune your debugger to the MCKO output clock of ARC Trace. This register exists only if you have included an offload Nexus interface during the build-time configuration.

Table 28-35 PTRN_GEN Field Descriptions

Field	Bits	Reset	Description
EN	[0]	0	**Enable pattern generation on Nexus Interface** ■ 0x1: pattern generation is enabled. ■ 0x0: pattern generation is disabled.
PT	[1]	0	**Generate specific bit pattern on the Nexus Interface.** **For eight-bit MDO Nexus interface** ■ 0x1: generate the following pattern: 0x55-> 0xAA-> 0x55-> 0xAA ■ 0x0: generate the following pattern: 0x00 -> 0xFF -> 0x00 -> 0xFF
Reserved	[31:2]	0	**Reserved. Read-as-zero; ignored on write.**

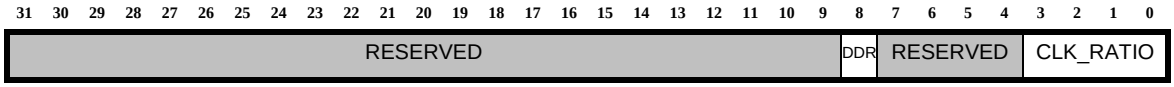
28.5.10 Nexus Clock Control Register, NEXUS_CLK_DIV

Address: 0x5A

Access: RW

Reset: 0x0

Figure 28-33 NEXUS_CLK_DIV Register



This register lets you configure the ratio between the system clock and the MCKO output clock of ARC Trace. System clock and ARC Trace clock should be phase aligned to support this clock ratios feature. This register is present only if ARC Trace is built with a Nexus interface.

Table 28-36 NEXUS_CLK_DIV Field Descriptions

Field	Bits	Reset	Description
CLK_RATIO	[3:0]	1	**Specifies the clock ratio between the system clock and the MCKO output clock** The allowed values are: 1,2,3,4,5,6,8 ■ 0x1: enable for clock ratio of 1 (default) ■ 0x2: enable for clock ratio of 2 ■ 0x3: enable for clock ratio of 3 ■ 0x4: enable for clock ratio of 4 ■ 0x5: enable for clock ratio of 5 ■ 0x6: enable for clock ratio of 6 ■ 0x7: enable for clock ratio of 8
Reserved	[7:4]	0	**Reserved. Read-as-zero; ignored on write.**
DDR (Double-Data Rate)	[8]	0	**Specifies if ARC Trace must offload trace messages at both the positive edge and negative edge of the Nexus clock** ■ 0x0: indicates that ARC Trace offloads trace messages only at the negative edge of the clock. ■ 0x1: indicates that ARC Trace offloads trace messages at both positive and negative edges of the clock. Note: When the CLK_RATIO == 1, ARC Trace offloads trace messages only on the negative edge of each clock cycle irrespective of the value of the Double-Data rate field.
Reserved	[31:9]	0	**Reserved. Read-as-zero; ignored on write.**

29 Cluster Network

29.1 Hardware/Software Interface

The cluster exposes a software programming interface to

- Read the cluster network (CLN) configuration
- Flush and invalidate the shared cache
- Program the shared cache and memory dimensions
- Program the address apertures for the shared memory, CCM, and NoC master ports
- Program the QoS (quality of service) parameters
- Access the safety features
- Access the security features
- Discovery, programming, and accessing of the performance monitoring counters
- Program and inspect the power state of the slave interfaces and SRAM banks

All CLN registers are 32 bits wide so that they can be accessed by both 32-bit cores and 64-bit cores.

Before accessing any cluster register, software must first verify the presence of the cluster registers. This is done by testing for a non-zero value in the `CLN_BCR` build configuration register at AUX address 0xCF.

29.1.1 Cluster Build Configuration Register, CLUSTER_BUILD

Address: 0xCF

Access: r

Figure 29-1 CLUSTER_BUILD Register

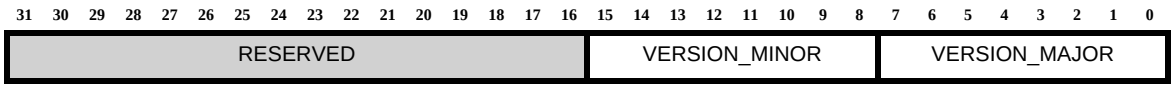


Table 29-1 CLUSTER_BUILD Field Description

Field	Range	Description
VERSION_MAJOR	[7:0]	Major version number of the cluster. Set to 0x20 for the first version. Among others the major version number defines the format and semantics of the AUX register space so it should be checked before proceeding with the access to other registers. All future versions of are guaranteed to have their major version number in this 8-bit field.
VERSION_MINOR	[15:8]	Minor version number of the cluster. Set to 0x0 for the first version. Minor versions are used to indicate different releases that are mutual compatible from the AUX register programming point of view (as long as they have the same major number).
RESERVED	[31:16]	Not used, read as zero.

The layout of the registers discussed below corresponds to `CLUSTER_BUILD.VERSION_MAJOR` == 8'd32. The programming interface of the cluster is quite extensive. The status and programming registers are for this reason not directly mapped into the AUX space, but can only be accessed indirectly by first writing the intended Network Register address to the `CLNR_ADDR` register in the common AUX space, and then accessing the corresponding register by either reading or writing the `CLNR_DATA` register. The cluster network guarantees that the order in which it receives AUX writes and reads from a particular ARC core, is exactly the order in which the AUX registers will be accessed. Therefore there is no need for software to check that an update to `CLNR_ADDR` has propagated before accessing the associated `CLNR_DATA` register. There are no ordering guarantees for AUX traffic originating from different ARC cores.

29.1.2 Register Overview

Table 29-2 Cluster Network Register Overview

Register Name	Address	Access	Description
CLUSTER_BUILD	0xCF	r	Network build configuration register

Table 29-2 Cluster Network Register Overview

Register Name	Address	Access	Description
CLUSTER_ID	0x298	r	Cluster identification number in the range 0..255. The number is obtained from input pins clusternum[7:0].
CLNR_ADDR	0x640	RW	Address of the CLN register to be accessed through CLNR_DATA. Available addresses are listed in Table 29-3. A physical copy of this register exists for every connected ARC processor. A read of this register returns the most-recently written address.
CLNR_DATA	0x641	RW	Proxy access to the register indicated by CLNR_ADDR. Accessing a non-existing register returns a bus error.
CLNR_DATA_NXT	0x642	RW	Proxy access to the register indicated by CLNR_ADDR. Accessing this register yields the same result as accessing CLNR_DATA, but it has the side effect of incrementing CLNR_ADDR by 1.
GLOBAL_CLK_GATE_DIS	0x9A9	RW	Clock gating control register
CLNR_BCR_0	0xF61	r	Cluster build configuration register #0. Same value as CLN_BCR_0.
CLNR_BCR_1	0xF62	r	Cluster build configuration register #1. Same value as CLN_BCR_1.
CLNR_BCR_2	0xF63	r	Cluster build configuration register #2. Same value as CLN_BCR_2.
CLNR_SCM_BCR_0	0xF64	r	SCM build configuration register #0. Returns 0 if SCM is not configured. Same value as CLN_SCM_BCR_0.
CLNR_SCM_BCR_1	0xF65	r	SCM build configuration register #1. Returns 0 if SCM is not configured. Same value as CLN_SCM_BCR_1.

In a multi-core configuration of the cluster, every connected master is allocated its own physical copy of `CLNR_ADDR`, even though each master is using the same auxiliary address to access its copy of `CLNR_ADDR`. This avoids race conditions if more than one master tries to access the shared CLNR space at about the same time. The `CLNR_ADDR` can be read which is useful in a save-restore sequence,

For example, when a debugger is preempting and later resuming software that may be accessing the CLN register space.

While Table 29-3 lists all defined registers, it is only possible to access registers that are configured according to the CLN configuration. Attempt to access registers associated with non-configured features results in a bus error when `CLNR_DATA` or `CLNR_DATA_NXT` is accessed.

Table 29-3 CLN Status and Control Register Overview

Register Name	CLN Address	Description
CLN_MST_NOC_0_BCR	0x0	NoC master port #0 build configuration register

Table 29-3 CLN Status and Control Register Overview (Continued)

Register Name	CLN Address	Description
...	...	...
CLN_MST_NOC_3_BCR	0x3	NoC master port #3 build configuration register
CLN_MST_PER_0_BCR	0x10	Peripheral master port #0 configuration register
CLN_MST_PER_1_BCR	0x11	Peripheral master port #1 configuration register
CLN_MST_CCM_0_BCR	0x20	CCM master port #0 build configuration register
...	...	...
CLN_MST_CCM_31_BCR	0x3F	CCM master port #31 build configuration register
CLN_SLV_0_BCR	0x40	Slave port #0 build configuration register
...	...	...
CLN_SLV_31_BCR	0x5F	Slave port #31 build configuration register
CLN_BCR_0	0x60	Cluster build configuration register #0
CLN_BCR_1	0x61	Cluster build configuration register #1
CLN_BCR_2	0x62	Cluster build configuration register #2
CLN_SCM_BCR_0	0x64	Shared Cache and Memory build configuration register 0
CLN_SCM_BCR_1	0x65	Shared Cache and Memory build configuration register 1
CLN_SHMEM_ADDR	0xC8	Lowest address of the shared memory aperture
CLN_SHMEM_SIZE	0xC9	Size of the shared memory aperture
CLN_SUPER_ADDR	0xCA	Lowest address of the superblock aperture
CLN_SUPER_SIZE	0xCB	Size of the superblock aperture
CLN_CACHE_ADDR_LO0	0xCC	Lowest address of a range for cache management [31:6]
CLN_CACHE_ADDR_LO1	0xCD	Lowest address of a range for cache management [51:32]
CLN_CACHE_ADDR_HI0	0xCE	Highest address of a range for cache management [31:6]
CLN_CACHE_ADDR_HI1	0xCF	Highest address of a range for cache management [51:32]
CLN_CACHE_CMD	0xD0	Cache operation command register
CLN_CACHE_STATUS	0xD1	Cache status and operation result
CLN_CACHE_ERR	0xD2	Cache writeback error status
CLN_CACHE_ERR_ADDR0	0xD3	Faulting address bits [31:0]
CLN_CACHE_ERR_ADDR1	0xD4	Faulting address bits [51:32]

Table 29-3 CLN Status and Control Register Overview (Continued)

Register Name	CLN Address	Description
CLN_MST_NOC_0_0_ADDR	0x124	Lowest address of the NoC master #0 aperture #0
CLN_MST_NOC_0_0_SIZE	0x125	Size of the NoC master #0 aperture #0
...	...	...
CLN_MST_NOC_3_0_ADDR	0x12A	Lowest address of the NoC master #3 aperture #0
CLN_MST_NOC_3_0_SIZE	0x12B	Size of the NoC master #3 aperture #0
CLN_PWR_STATUS_0	0x17C	Power-on/off status of ARC cores
CLN_PWR_STATUS_1	0x17D	Power-on/off status of master ports
CLN_PWR_STATUS_2	0x17E	Power-on/off status of slave ports
CLN_QOS_CTL	0x180	QOS control common to all resource domains.
CLN_QOS_RDOM0_FOOTPRINT	0x18A	QoS cache footprint control for resource domain #0
CLN_QOS_RDOM1_FOOTPRINT	0x192	QoS cache footprint control for resource domain #1
...	...	...
CLN_QOS_RDOM7_FOOTPRINT	0x1C2	QoS cache footprint control for resource domain #7
CLN_MST_CCM_0_0_ADDR	0x300	Lowest address of the CCM master #0 aperture #0
CLN_MST_CCM_0_0_SIZE	0x301	Size of the CCM master #0 aperture #0
CLN_MST_CCM_0_1_ADDR	0x302	Lowest address of the CCM master #0 aperture #1
CLN_MST_CCM_0_1_SIZE	0x303	Size of the CCM master #0 aperture #1
CLN_MST_CCM_0_2_ADDR	0x304	Lowest address of the CCM master #0 aperture #2
CLN_MST_CCM_0_2_SIZE	0x305	Size of the CCM master #0 aperture #0
CLN_MST_CCM_0_3_ADDR	0x306	Lowest address of the CCM master #0 aperture #3
CLN_MST_CCM_0_3_SIZE	0x307	Size of the CCM master #0 aperture #1
CLN_MST_CCM_1_0_ADDR	0x308	Lowest address of the CCM master #1 aperture #0
CLN_MST_CCM_1_0_SIZE	0x309	Size of the CCM master #1 aperture #0
CLN_MST_CCM_1_1_ADDR	0x30A	Lowest address of the CCM master #1 aperture #1
...	...	...
CLN_MST_CCM_31_3_ADDR	0x3FE	Lowest address of the CCM master #31 aperture #3
CLN_MST_CCM_31_3_SIZE	0x3FF	Size of the CCM master #31 aperture #3
CLN_WR_ERR	0x680	Posted write error status

Table 29-3 CLN Status and Control Register Overview (Continued)

Register Name	CLN Address	Description
CLN_WR_ERR_ADDR0	0x681	Posted write error address bits [31:0]
CLN_WR_ERR_ADDR1	0x682	Posted write error address bits [51:32]
CLN_ATTN	0x690	Status of 'attn' interrupt signal.
CLN_MST_CCM_0_UAUX	0x900	Master CCM port #0 UAUX aperture configuration
...	...	...
CLN_MST_CCM_31_UAUX	0x91F	Master CCM port #31 UAUX aperture configuration
CLN_MST_CCM_0_XSPC0	0x920	Master CCM port #0 XSPC0 aperture configuration
...	...	...
CLN_MST_CCM_31_XSPC0	0x93F	Master CCM port #31 XSPC0 aperture configuration
CLN_MST_CCM_0_XSPC1	0x940	Master CCM port #0 XSPC1 aperture configuration
...	...	...
CLN_MST_CCM_31_XSPC1	0x95F	Master CCM port #31 XSPC1 aperture configuration
CLN_DBNK_ECC_CTRL	0x960	Data bank ECC control
CLN_TBNK_ECC_CTRL	0x961	Data tag bank ECC control
CLN_STAG_ECC_CTRL	0x962	Shadow tag memory ECC control
CLN_CDMA_ECC_CTRL	0x963	DMA Descriptor ECC control
CLN_MST_NOC_0_1_ADDR	0xA00	Lowest address of the NoC master #0 aperture #1
CLN_MST_NOC_0_1_SIZE	0xA01	Size of the NoC master #0 aperture #1
CLN_MST_NOC_0_2_ADDR	0xA02	Lowest address of the NoC master #0 aperture #2
CLN_MST_NOC_0_2_SIZE	0xA03	Size of the NoC master #0 aperture #2
...	...	...
CLN_MST_NOC_0_7_ADDR	0xA0C	Lowest address of the NoC master #0 aperture #7
CLN_MST_NOC_0_7_SIZE	0xA0D	Size of the NoC master #0 aperture #7
CLN_MST_NOC_1_1_ADDR	0xA10	Lowest address of the NoC master #1 aperture #1
CLN_MST_NOC_1_1_SIZE	0xA11	Size of the NoC master #1 aperture #1
...	...	...
CLN_MST_NOC_3_7_ADDR	0xA3C	Lowest address of the NoC master #3 aperture #7
CLN_MST_NOC_3_7_SIZE	0xA3D	Size of the NoC master #3 aperture #7

Table 29-3 CLN Status and Control Register Overview (Continued)

Register Name	CLN Address	Description
CLN_PER0_BASE	0xA80	Lowest address of the PER0 peripheral aperture
CLN_PER0_SIZE	0xA81	Size of the PER0 peripheral aperture
CLN_PER1_BASE	0xA82	Lowest address of the PER1 peripheral aperture
CLN_PER1_SIZE	0xA83	Size of the PER1 peripheral aperture
CLN_SLV_0_QOS	0xAC0	QoS resource domain of transactions from slave port #0
...	...	...
CLN_SLV_31_QOS	0xADF	QoS resource domain of transactions from slave port #31
CLN_SLV_0_ARCT	0xAE0	ARC Trace control register for ARC core #0
...	...	...
CLN_SLV_11_ARCT	0xAEB	ARC Trace control register for ARC core #11
CLN_SOFTRESET_CORES	0xB00	Information on soft reset status (global and ARC cores)
CLN_SOFTRESET_DEV	0xB01	Information on soft reset status (devices)
Cluster performance counter registers	0xC00 to 0xFFF	Register address space reserved for cluster performance counters.

In the following subsections each of these CLNR registers is described in detail. Fields of a register can be read or written in one of the following modes

- R: readable
- RAZ: read as zero
- W: writable
- IOW: ignored on write

Many registers have a reset value specified, in the tables below this is indicated as @reset=value. If no @reset value has been specified then the register is usually read-only and immutable.

Any attempt to access a non-existing CLN register results in an error response for the IBP AUX transaction. If the AUX transaction is associated with a committed `lr` or `sr` instruction then the associated exception will not be precise. Therefore it is recommended that the core that issues the AUX-space read or write, verifies the validity of the register address before submitting the IBP transaction to the cluster network.

In several cases, and when `CFG_ACR`=1, it is possible to write a value to an active configuration register (ACR) with the effect of down-configuring the available hardware resources. The primary purpose of this feature is to provide a means to explore cost-effective configurations on, for example, a test chip that has been configured with abundant resources. In some cases, down configurations can also reduce power consumption if combined with certain PSO (power shut-off) features.

29.1.2.1 NoC Master Port #0 Build Configuration Register,CLN_MST_NOC_*0*_BCR

CLNR Address: 0x0

Access: R

Figure 29-2 CLN_MST_NOC_*0*_BCR Register

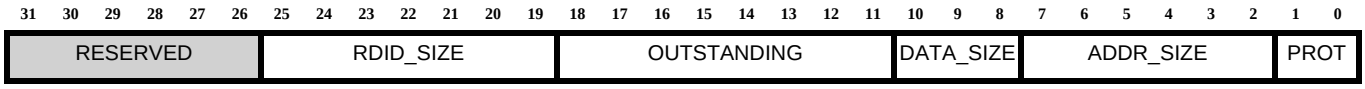


Table 29-4 CLN_MST_NOC_*0*_BCR Field Descriptions

Field	Bit	Access	Description
PROT	[1:0]	R, IOW	Protocol: 00 (IBPv3), 01 (AXI4), 10 (ACE) 11 (ACE+DVM).
ADDR_SIZE	[7:2]	R, IOW	#address bits on a NoC master port.
DATA_SIZE	[10:8]	R, IOW	#data bits on the NoC read/write channels: 64 bits (001), 128 bits (010), 512 bits (100). Other values reserved.
OUTSTANDING	[18:11]	R, IOW	Max #outstanding transactions on a NoC master (CFG_MST_NOC_OUTSTANDING[*i*]).
RDID_SIZE	[25:19]	R, IOW	# RDID bits in the NOC master interface.
Reserved	[31:26]	RAZ, IOW	Reserved.

29.1.2.2 NoC Master Port #1 Build Configuration Register, CLN_MST_NOC_*1*_BCR

CLNR Address: 0x1

Access: R

Figure 29-3 CLN_MST_NOC_*1*_BCR Register

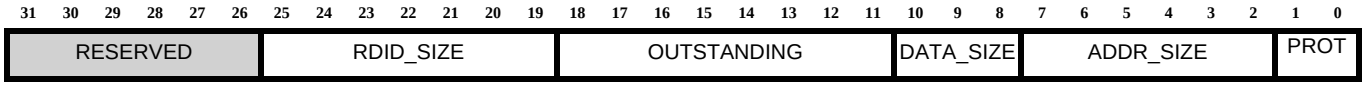


Table 29-5 CLN_MST_NOC_*1*_BCR Field Description

Field	Bit	Access	Description
PROT	[1:0]	R, IOW	Protocol: 00 (IBPv3), 01 (AXI4), 10 (ACE) 11 (ACE+DVM).
ADDR_SIZE	[7:2]	R, IOW	#address bits on a NoC master port.
DATA_SIZE	[10:8]	R, IOW	#data bits on the NoC read/write channels: 64 bits (001), 128 bits (010), 512 bits (100). Other values reserved.
OUTSTANDING	[18:11]	R, IOW	Max #outstanding transactions on a NoC master (CFG_MST_NOC_OUTSTANDING[*i*]).
RDID_SIZE	[25:19]	R, IOW	# RDID bits in the NOC master interface.
Reserved	[31:26]	RAZ, IOW	Reserved.

29.1.2.3 NoC Master Port #2 Build Configuration Register, CLN_MST_NOC_*2*_BCR

CLNR Address: 0x2

Access: R

Figure 29-4 CLN_MST_NOC_*2*_BCR Register

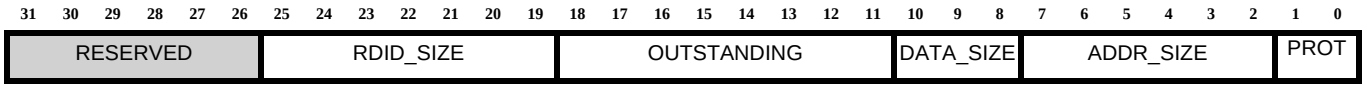


Table 29-6 CLN_MST_NOC_*2*_BCR Field Description

Field	Bit	Access	Description
PROT	[1:0]	R, IOW	Protocol: 00 (IBPv3), 01 (AXI4), 10 (ACE) 11 (ACE+DVM).
ADDR_SIZE	[7:2]	R, IOW	#address bits on a NoC master port.
DATA_SIZE	[10:8]	R, IOW	#data bits on the NoC read/write channels: 64 bits (001), 128 bits (010), 512 bits (100). Other values reserved.
OUTSTANDING	[18:11]	R, IOW	Max #outstanding transactions on a NoC master (CFG_MST_NOC_OUTSTANDING[*i*]).
RDID_SIZE	[25:19]	R, IOW	# RDID bits in the NOC master interface.
Reserved	[31:26]	RAZ, IOW	Reserved.

29.1.2.4 NoC Master Port #3 Build Configuration Register, CLN_MST_NOC_*3*_BCR

CLNR Address: 0x3

Access: R

Figure 29-5 CLN_MST_NOC_*3*_BCR Register

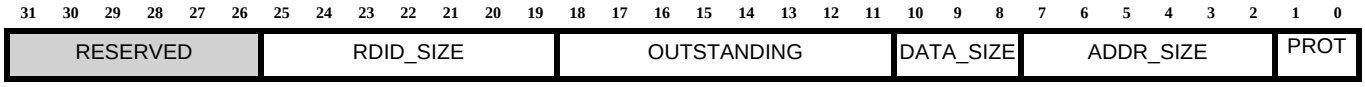


Table 29-7 CLN_MST_NOC_*3*_BCR Field Description

Field	Bit	Access	Description
PROT	[1:0]	R, IOW	Protocol: 00 (IBPv3), 01 (AXI4), 10 (ACE) 11 (ACE+DVM).
ADDR_SIZE	[7:2]	R, IOW	#address bits on a NoC master port.
DATA_SIZE	[10:8]	R, IOW	#data bits on the NoC read/write channels: 164 bits (001), 28 bits (010), 512 bits (100). Other values reserved.
OUTSTANDING	[18:11]	R, IOW	Max #outstanding transactions on a NoC master (CFG_MST_NOC_OUTSTANDING[*i*]).
RDID_SIZE	[25:19]	R, IOW	# RDID bits in the NOC master interface.
Reserved	[31:26]	RAZ, IOW	Reserved

29.1.2.5 NoC Master Port #0 Build Configuration Register, CLN_MST_PER_*0*_BCR

CLNR Address: 0x16

Access: R

Figure 29-6 CLN_MST_PER_*0*_BCR Register

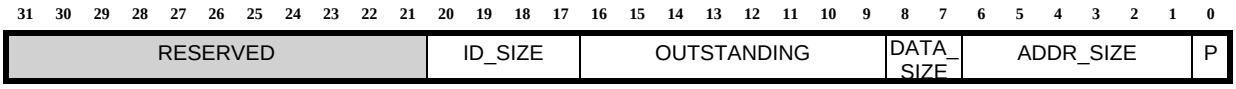


Table 29-8 CLN_MST_PER_0_BCR Field Description

Field	Bit	Access	Description
PROTOCOL(P)	[0]	R, IOW	Protocol: ■ `0x0`: AXI ■ `0x1`: AHB-Lite
ADDR_SIZE	[6:1]	R, IOW	Address bits on a peripheral master port.
DATA_SIZE	[8:7]	R, IOW	Data bits on the peripheral read/write channels: ■ `0x0`: 32 bit ■ `0x1`: 64 bit
OUTSTANDING	[16:9]	R, IOW	Maximum outstanding transactions on the peripheral master (CFG_MST_PER_OUTSTANDING[*i*]).
ID_SIZE	[20:17]	R, IOW	ID bits in the peripheral master interface.
Reserved	[31:21]	RAZ, IOW	Reserved. Read as zero and ignore on write

29.1.2.6 Peripheral Master Port #1 Build Configuration Register, CLN_MST_PER_*1*_BCR

CLNR Address: 0x17

Access: R

Figure 29-7 CLN_MST_PER_*1*_BCR Register

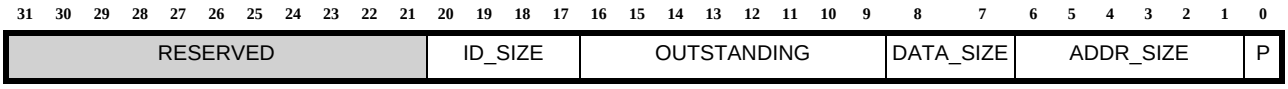


Table 29-9 CLN_MST_PER_1_BCR Field Description

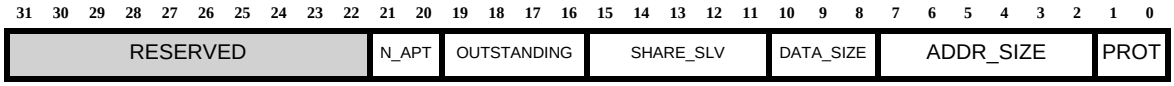
Field	Bit	Access	Description
PROTOCOL(P)	[0]	R, IOW	Protocol: ■ `0x0`: AXI ■ `0x1`: AHB-Lite
ADDR_SIZE	[6:1]	R, IOW	Address bits on a peripheral master port.
DATA_SIZE	[8:7]	R, IOW	Data bits on the peripheral read/write channels: ■ `0x0`: 32 bit ■ `0x1`: 64 bit
OUTSTANDING	[16:9]	R, IOW	Maximum outstanding transactions on the peripheral master (CFG_MST_PER_OUTSTANDING[*i*]).
ID_SIZE	[20:17]	R, IOW	ID bits in the peripheral master interface.
Reserved	[31:21]	RAZ, IOW	Reserved. Read as zero and ignore on write

29.1.2.7 CCM Master Port #i Build Configuration Registers, CLN_MST_CCM_*i*_BCR

CLNR Address: 0x20 to 0x3F

Access: R

Figure 29-8 CLN_MST_CCM_*i*_BCR Registers



The value for field X is the build-time configuration parameter `CFG_MST_CCM_X[i]`.

Table 29-10 CLN_MST_CCM_*i*_BCR Field Description

Field	Bit	Access	Description
PROT	[1:0]	R, IOW	Protocol: 00 (IBPv3), 01 (AXI4), 10 (ARCDMI). Other values reserved.
ADDR_SIZE	[7:2]	R, IOW	#address bits on a CCM master port.
DATA_SIZE	[10:8]	R, IOW	#data bits on the CCM read/write channels: 32 bits (000), 64 bits (001), 128 bits (010), 256 bits (011), 512 bits (100). Other values reserved.
SHARE_SLV	[15:11]	R, IOW	R/W channel sharing with slave interface SLV. Read as 5'b11111 if the R/W channels are not shared with any slave interface.
OUTSTANDING	[19:16]	R, IOW	Max #outstanding transactions on a CCM master. (CFG_MST_CCM_OUTSTANDING[*i*]). The value 16 is encoded as '0000'.
N_APT	[21:20]	R, IOW	#memory apertures associated with this CCM port (CFG_MST_CCM[*i*].mem_n_apt). The value '4' is encoded as '00'. Each aperture *k* is defined by the registers CLN_MST_CCM_*i*_*k*_ADDR and CLN_MST_CCM_*i*_*k*_SIZE.
Reserved	[31:22]	RAZ, IOW	Reserved

29.1.2.8 CLN_MST_CCM_*i*_UAUX Registers

CLNR Address: 0x900 to 0x91F

Access: R

Build configuration for the Master CCM port #i UAUX aperture configuration

Figure 29-9 CLN_MST_CCM_*i*_UAUX

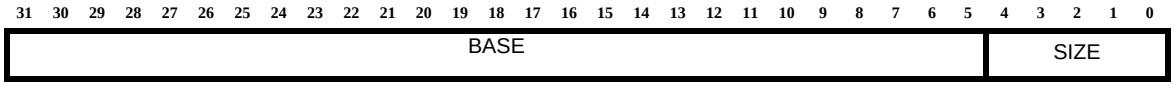


Table 29-11 CLN_MST_CCM_*i*_UAUX Field Description

Field	Bit	Access	Description
SIZE	[5:0]	R, IOW	Aperture size is 2^{4+SIZE}. Read as SIZE=0 if aperture is non-existent.
BASE	[31:5]	R, IOW	Aperture base address bits [31:5], with address bits [4:0] assumed all 0. Must be a multiple of aperture size.

29.1.2.9 CLN_MST_CCM_*i*_XSPC0 Registers

CLNR Address: 0x920 to 0x93F

Access: R

Build configuration for the master CCM port #i XSPC0 aperture configuration.

Figure 29-10 CLN_MST_CCM_*i*_XSPC0

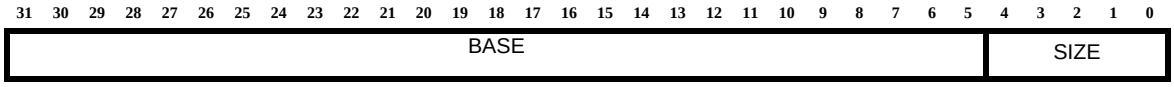


Table 29-12 CLN_MST_CCM_*i*_XSPC0 Field Description

Field	Bit	Access	Description
SIZE	[5:0]	R, IOW	Aperture size is 2^{4+SIZE}. Read as SIZE=0 if aperture is non-existent.
BASE	[31:5]	R, IOW	Aperture base address bits [31:5], with address bits [4:0] assumed all 0. Must be a multiple of aperture size.

29.1.2.10 CLN_MST_CCM_*i*_XSPC1 Registers

CLNR Address: 0x940 to 0x95F

Access: R

Build configuration for the master CCM port #i XSPC1 aperture configuration

Figure 29-11 CLN_MST_CCM_*i*_XSPC1

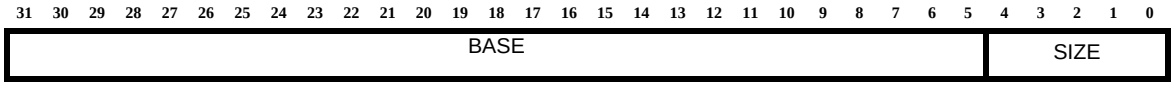


Table 29-13 CLN_MST_CCM_*i*_XPSC1 Field Description

Field	Bit	Access	Description
SIZE	[5:0]	R, IOW	Aperture size is 2^{4+SIZE}. Read as SIZE=0 if aperture is non-existent.
BASE	[31:5]	R, IOW	Aperture base address bits [31:5], with address bits [4:0] assumed all 0. Must be a multiple of aperture size.

29.1.2.11 Slave Port #i Build Configuration Registers, CLN_SLV_*i*_BCR

CLNR Address: 0x40 to 0x5F

Access: R

Figure 29-12 CLN_SLV_*i*_BCR Registers

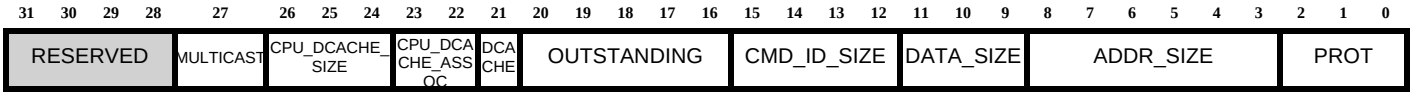


Table 29-14 CLN_SLV_*i*_BCR Field Description

Field	Bit	Access	Description
PROT	[2:0]	R, IOW	Bus protocol: 000 (IBP), 001 (AXI4), 010 (Ace-Lite), 100 (APB0/trace), 101 (APB1/safety), 110 (APB2/trace+safety).
ADDR_SIZE	[8:3]	R, IOW	#address bits on a slave port (64 encoded as 0).
DATA_SIZE	[11:9]	R, IOW	#data bits on the slave read/write channels: 32 bits (000), 64 bits (001), 128 bits (010), 256 bits (011), 512 bits (100). Other values reserved.
CMD_ID_SIZE	[15:12]	R, IOW	#bits in CMD_ID, with 16 encoded as 0000.
OUTSTANDING	[20:16]	R, IOW	#outstanding IBPv3 transactions supported on a slave interface: CFG_SLV_CMD_OUTSTANDING[*i*]. The value 32 is encoded as 00000.
DCACHE	[21]	R, IOW	1 if this slave is connected to an ARC cached master. 0 if it is connected to an uncached device.
CPU_DCACHE_ASSOC	[23:22]	R, IOW	Read as zero if DCACHE is 0. Otherwise this is the associativity of the L1 data cache: 2 (01), 4 (10), 8 (11).
CPU_DCACHE_SIZE	[26:24]	R, IOW	Read as zero if DCACHE is 0. Otherwise this represents the L1 data cache size: 4 kB (000), 8 kB (001), 16 kB (010), 32 kB (011), 64 kB (100).
MULTICAST	[27]	R, IOW	Support for multicast: 1 (yes), 0 (no).
RESERVED	[31:28]	RAZ, IOW	Reserved.

29.1.2.12 Cluster Build Configuration Register #0, CLN_BCR_0

CLNR Address: 0x96

Access: r

This is build configuration register #0. It is also available directly in the AUX address space as `CLNR_BCR_0`.

Figure 29-13 CLN_BCR_0 Register

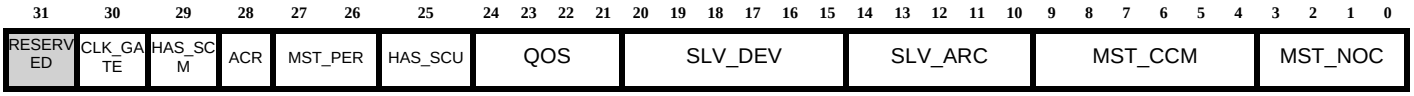


Table 29-15 CLN_BCR_0 Field Description

Field	Bit	Access	Description
MST_NOC	[3:0]	R, IOW	Number of NoC master ports. Equals CFG_MST_NOC_COUNT.
MST_CCM	[9:4]	R, IOW	Number of CCM master ports. Equals CFG_MST_CCM_COUNT.
SLV_ARC	[14:10]	R, IOW	Number of ARC slave ports. Equals CFG_SLV_ARC_COUNT.
SLV_DEV	[20:15]	R, IOW	Number of device slave ports. Equals CFG_SLV_DEV_COUNT. The presence or absence of CDMA does not affect this number.
QOS	[24:21]	R, IOW	#QoS resource domains: CFG_QOS. If 0 then QoS is not available.
HAS_SCU	[25]	R, IOW	SCU present (1) or not (0).
MST_PER	[27:26]	R, IOW	Number of peripheral master ports. Equals CFG_MST_PER_COUNT.
ACR	[28]	R, IOW	1 if we have ACR_* registers, 0 if not.
HAS_SCM	[29]	R, IOW	Shared Cache and Memory present (1) or not (0).
CLK_GATE	[30]	R, IOW	If 1 then clock gating and the GLOBAL_CLK_GATE_DIS register are configured implemented.
RESERVED	[31]	RAZ, IOW	Reserved.

29.1.2.13 Cluster Build Configuration Register #1, CLN_BCR_1

CLNR Address: 0x97

Access: r

This is build configuration register #01. It is also available directly in the AUX address space as CLNR_BCR_1.

Figure 29-14 CLN_BCR_1 Register

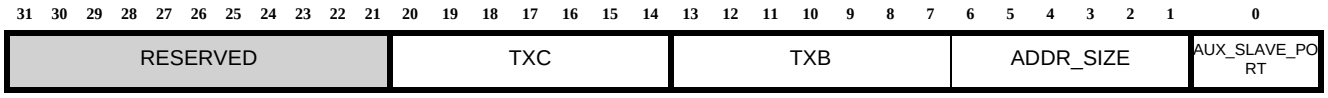


Table 29-16 CLN_BCR_1 Field Description

Field	Bit	Access	Description
AUX_SLAVE_PORT	[0]	R, IOW	if 1 then an AHB slave port is configured to enable external access to the cluster network AUX registers: CFG_AUX_SLAVE_PORT.
ADDR_SIZE	[6:1]	R, IOW	Physical address size used internally by the cluster network: CFG_ADDR_SIZE.
TXB	[13:7]	R, IOW	The #transaction buffers: CFG_TXB. The number 128 is represented as 0.
TXC	[20:14]	R, IOW	The #transaction controller: CFG_TXC. The number 128 is represented as 0.
RESERVED	[31:21]	RAZ, IOW	Reserved.

29.1.2.14 Cluster Build Configuration Register #2, CLN_BCR_2

CLNR Address: 0x98

Access: r

This is build configuration register #02. It is also available directly in the AUX address space as `CLNR_BCR_2`.

Figure 29-15 CLN_BCR_2 Register

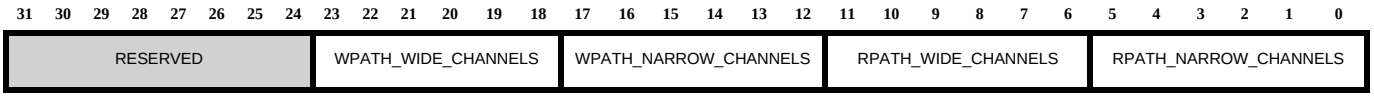


Table 29-17 CLN_BCR_1 Field Description

Field	Bit	Access	Description
RPATH_NARROW_CHANNELS	[5:0]	R, IOW	The #concurrent transactions through the narrow rpath: CFG_RPATH_NARROW_CHANNELS.
RPATH_WIDE_CHANNELS	[11:6]	R, IOW	The #concurrent transactions through the wide rpath: CFG_RPATH_WIDE_CHANNELS.
WPATH_NARROW_CHANNELS	[17:12]	R, IOW	The #concurrent transactions through the narrow wpath: CFG_WPATH_NARROW_CHANNELS.
WPATH_WIDE_CHANNELS	[23:18]	R, IOW	The #concurrent transactions through the wide wpath: CFG_WPATH_WIDE_CHANNELS.
RESERVED	[31:24]	RAZ, IOW	Reserved.

29.1.2.15 Shared Cache and Memory (SCM) Configuration Register 0, CLN_SCM_BCR_0

CLNR Address: 0x100

Access: r

This register is also available directly in the AUX address space as `CLNR_SCM_BCR_0`.

Figure 29-16 CLN_SCM_BCR_0 Register

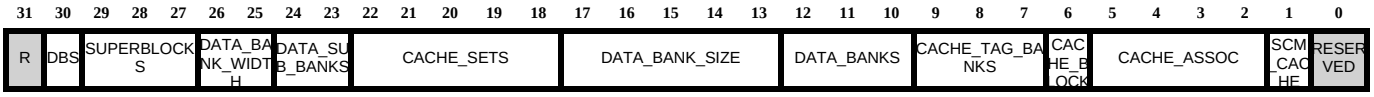


Table 29-18 CLN_SCM_BCR_0 Field Description

Field	Bit	Access	Description
RESERVED	[0]	RAZ, IOW	Reserved.
SCM_CACHE	[1]	R, IOW	Shared cache present (1) or not (0).
CACHE_ASSOC	[5:2]	R, IOW	Configured associativity of the shared cache: CFG_SCM_CACHE_ASSOC. The value 16 is represented by 0000.
CACHE_BLOCK_SIZE	[6]	R, IOW	Shared cache block size: 64 bytes (1), other values reserved. Equals CFG_SCM_CACHE_BLOCK_SIZE.
CACHE_TAG_BANKS	[9:7]	R, IOW	Configured #banks in the cache tag memory. Encoding: 2 (001), 4 (010), 8 (011), 16 (100). Other values reserved. Equals CFG_SCM_CACHE_TAG_BANKS.
DATA_BANKS	[12:10]	R, IOW	Configured #banks in the data memory. These banks are used both for shared memory and for the shared cache. Encoding: 2 (001), 4 (010), 8 (011), 16 (100), 32 (101). Other values reserved. Equals CFG_SCM_DATA_BANKS.
DATA_BANK_SIZE	[17:13]	R, IOW	Log2 of the size of a data bank, expressed in kB. For example: 00000 (1 kB), 00111 (128 kB), 01101 (8 MB). This equals Log2 (CFG_SCM_DATA_SIZE (in kB) / CFG_SCM_DATA_BANKS).
CACHE_SETS	[22:18]	R, IOW	Log2 of the number of sets in the shared cache: *a* / (CACHE_ASSOC * CACHE_BLOCK_SIZE), where *a* is (DATA_BANKS * DATA_BANK_SIZE – CLN_SHMEM_SIZE) rounded to the next power of 2.
DATA_SUB_BANKS	[24:23]	R, IOW	#SRAMs per data bank: 2 (01) or 4 (10). Other values reserved. Equivalent to CFG_SCM_DATA_SUB_BANKS.
DATA_BANK_WIDTH	[26:25]	R, IOW	Data bank internal channel width: 128 bits (00), 256 bits (01), 512 bits (10). Equivalent to CFG_SCM_DATA_BANK_WIDTH.

Table 29-18 CLN_SCM_BCR_0 Field Description

Field	Bit	Access	Description
SUPERBLOCKS	[29:27]	R, IOW	000: no superblocks, 001: 128-byte superblocks, 010: 256-byte superblocks, 011: 512 byte superblocks. Other values reserved.
DBS (DATA_BANK_SIZE_75)	[30]	R, IOW	The actual data bank size is 75% of what is indicated in the DATA_BANK_SIZE field. For example, with DATA_BANK_SIZE = 01010 and DATA_BANK_SIZE_75 = 1 we have a 768 kB data bank.
RESERVED	[31:29]	RAZ, IOW	Reserved.

29.1.2.16 Shared Cache and Memory (SCM) Configuration Register 1, CLN_SCM_BCR_1

CLNR Address: 0x101

Access: r

This register is also available directly in the AUX address space as `CLNR_SCM_BCR_1`.

Figure 29-17 CLN_SCM_BCR_1 Register

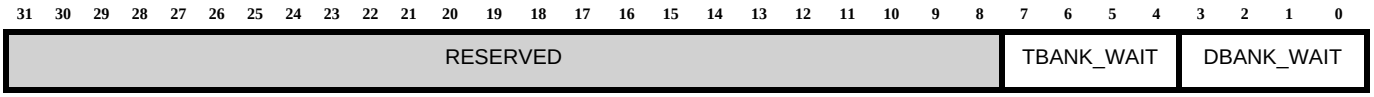


Table 29-19 CLN_SCM_BCR_1 Field Description

Field	Bit	Access	Description
DBANK_WAIT	[3:0]	R, IOW	Reserved.
TBANK_WAIT	[7:4]	R, IOW	Shared cache present (1) or not (0).
RESERVED	[31:8]	RAZ, IOW	Configured associativity of the shared cache: CFG_SCM_CACHE_ASSOC. The value 16 is represented by 0000.

29.2 Modifying the Cluster Network Configuration

It is possible to change the configuration of the cluster network. The primary purpose for this is to enable experimentation with smaller hardware configurations, typically to see if a low-cost version can be used and still have acceptable performance. The layout of the Active Configuration Registers (ACRs) is intentionally similar to the layout of the read-only Build Configuration Registers (BCRs).

29.2.1 Shared Cache and Memory Active Configuration Register 1, CLN_SCM_ACR_1

CLNR Address: 0x2F9

Access: RW

This register enables modification of the timing of the data banks and tag banks. This register helps to reduce the number of dbank wait cycles when the dbank clock frequency is lowered, typically to save power. At lower clock frequencies and different Vdd operating points the number of required wait cycles can change and that is when system software can program `CLN_SCM_ACR_1.DBANK_WAIT != CLN_SCM_BCR_1.DBANK_WAIT`. When system software does this, it must limit itself to operating points that have been validated through static timing analysis.

Figure 29-18 CLN_SCM_ACR_1 Register

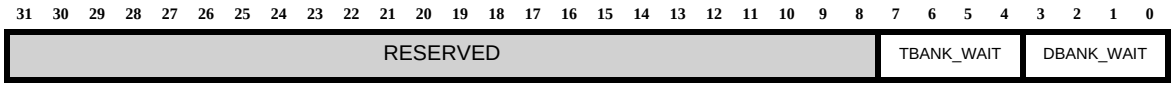


Table 29-20 CLN_SCM_ACR_1 Field Description

Field	Bit	Access	Description
DBANK_WAIT	[3:0]	R, W	Number of wait cycles used for data bank access. Zero means single-cycle access. @reset= CFG_SCM_DWAIT_RESET.
TBANK_WAIT	[7:4]	R, W	Number of wait cycles used for tag bank access. Zero means single-cycle access. @reset= CFG_SCM_TWAIT_RESET.
RESERVED	[31:8]	RAZ, IOW	Reserved.

Note The CLN_SCM_ACR_1.DBANK_WAIT filed should not be greater than CLN_SCM_BCR_1.DBANK_WAIT field. Similarly, the CLN_SCM_ACR_1.TBANK_WAIT field should not be greater than CLN_SCM_BCR_1.TBANK_WAIT field.

29.3 Programming the Shared Memory Aperture

The shared memory address aperture is disabled at reset and can be enabled by writing to the registers CLN_SHMEM_ADDR and CLN_SHMEM_SIZE. In case of the SHMEM aperture overlaps with a master port aperture, this is not considered an error and the SHMEM aperture takes priority. For more information, see "Programming Master Port Address Apertures" on page 1192.

29.3.1 Shared Memory Aperture Lowest Address Register, CLN_SHMEM_ADDR

CLNR Address: 0xC8

Access: RW

Figure 29-19 CLN_SHMEM_ADDR Register

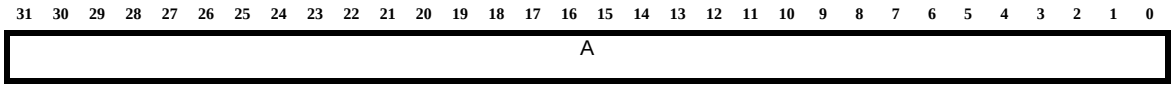


Table 29-21 CLN_SHMEM_ADDR Field Description

Field	Bit	Access	Description
A	[31:0]	R, W	Bits [51:20] of the shared memory base address. Bits [19:0] are always zero, i.e. the base address is always aligned at a 1 MB boundary. In addition, the base address must be aligned with the next-power-of-2 size of the shmem aperture, see CLN_SHMEM_SIZE. @reset=0.

29.3.2 Shared Memory Aperture Size Register, CLN_SHMEM_SIZE

CLNR Address: 0xC9

Access: RW

Figure 29-20 CLN_SHMEM_SIZE Register

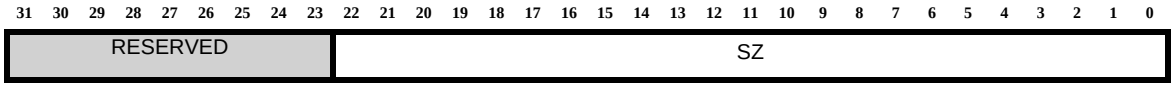


Table 29-22 CLN_SHMEM_SIZE Field Description

Field	Bit	Access	Description
SZ	[22:0]	R, W	The size of the shared memory aperture in 64 byte blocks. For sizes > 1 MB the following constraints hold for the base address CLN_SHMEM_ADDR.A: ■ SZ in <1 MB...2 MB] → align base address at 2 MB ■ SZ in <2 MB...4 MB] → align base address at 4 MB ■ SZ in <4 MB...8 MB] → align base address at 8 MB ■ SZ ini <8 MB...16 MB] → align base address at 16 MB ■ Etc. When SZ is equal to CFG_SCM_DATA_SIZE/64, the shared cache is effectively disabled. @reset=0.
RESERVED	[31:23]	RAZ, IOW	Reserved.

29.4 Flushing and Invalidating the Shared Cache

29.4.1 Lowest Address of a Range for CLN_CACHE_ADDR_LO0 [31:6] Register

CLNR Address: 0xCC

Access: RW

Figure 29-21 CLN_CACHE_ADDR_LO0 Register

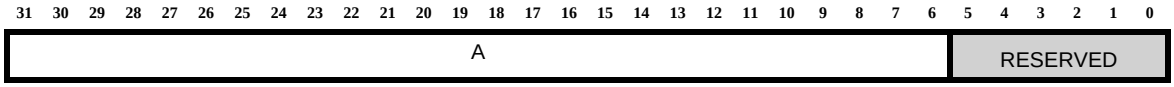


Table 29-23 CLN_CACHE_ADDR_LO0 Register

Field	Bit	Access	Description
RESERVED	[5:0]	RAZ, IOW	Reserved.
A	[31:6]	R, W	Address bits [31:6] of the low address associated with CLN_CACHE_CMD. @reset=0.

29.4.2 Lowest Address of a Range for CLN_CACHE_ADDR_LO1 [51:32] Register

CLNR Address: 0xCD

Access: RW

Figure 29-22 CLN_CACHE_ADDR_LO1 Register

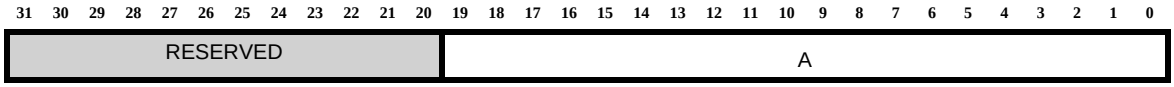


Table 29-24 CLN_CACHE_ADDR_LO1 Field Description

Field	Bit	Access	Description
A	[19:0]	R, W	Address bits [51:32] of the low address associated with CLN_CACHE_CMD. The upper (52 - CFG_ADDR_SIZE) bits of this field are ignored by CLN_CACHE_CMD operations. @reset=0
RESERVED	[31:20]	RAZ, IOW	Reserved.

29.4.3 Highest Address of a Range for CLN_CACHE_ADDR_HI0 [31:0] Register

CLNR Address: 0xCE

Access: RW

Figure 29-23 CLN_CACHE_ADDR_HI0 Register

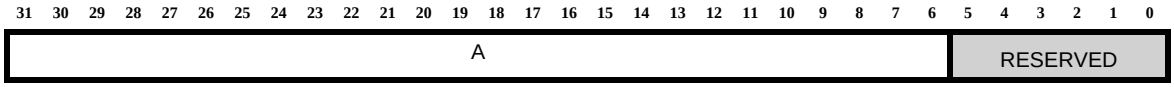


Table 29-25 CLN_CACHE_ADDR_HI0 Field Description

Field	Bit	Access	Description
RESERVED	[5:0]	RAZ, IOW	Reserved.
A	[31:6]	R, W	Address bits [31:6] of the high address associated with CLN_CACHE_CMD. @reset=0.

29.4.4 Highest Address of a Range for CLN_CACHE_ADDR_HI1 [51:32] Register

CLNR Address: 0xCF

Access: RW

Figure 29-24 CLN_CACHE_ADDR_HI1 Register

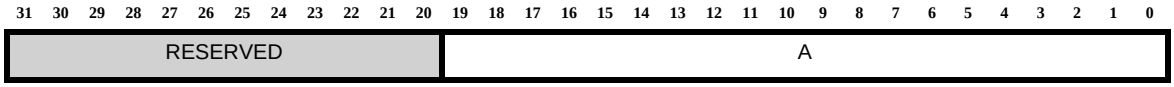


Table 29-26 CLN_CACHE_ADDR_HI1 Field Description

Field	Bit	Access	Description
A	[19:0]	R, W	Address bits [51:32] of the high address associated with CLN_CACHE_CMD. The upper (52 - CFG_ADDR_SIZE) bits of this field are ignored by CLN_CACHE_CMD operations. @reset=0.
RESERVED	[31:20]	RAZ, IOW	Reserved.

29.4.5 Cache Operation Command Register, CLN_CACHE_CMD

CLNR Address: 0xD0

Access: RW

Figure 29-25 CLN_CACHE_CMD Register

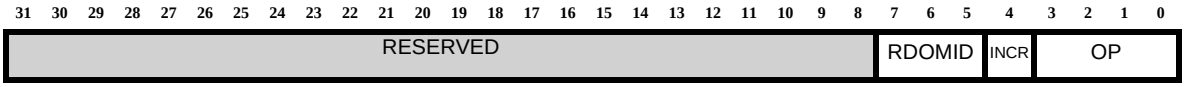


Table 29-27 CLN_CACHE_CMD Field Description

Field	Bit	Access	Description
OP	[3:0]	RAZ, W	Cache operation. These cache operations do not block or inhibit progress of normal cluster network and shared cache and memory operations. In the case of region based commands (see below) all the normal rules with respect to concurrency and multiple outstanding requests apply. Also the QoS rules apply, where the resource domain of the cache operations is taken from CACHE_CMD.RDOMID. Unless specified otherwise, cache operations execute irrespective of the value of CLN_CACHE_STATUS.ENABLE ■ NOP (0000): do nothing. ■ LOOKUP (0001): Lookup the line with address {CACHE_ADDR_LO1.A, CACHE_ADDR_LO0} and put the result in CACHE_STATUS.MOESIP. If it is not 'I' then update CACHE_STATUS.INDEX and CACHE_STATUS.WAY, otherwise leave these unchanged. ■ PROBE (0010): Lookup the line at (CACHE_STATUS.INDEX, CACHE_STATUS.WAY) and put the line address in {CACHE_ADDR_LO1.A, CACHE_ADDR_LO0}. Also put the MOESI state in CACHE_STATUS.MOESIP. ■ INDEX_INVALIDATE (0101): Set MOESIP state of line (CACHE_STATUS.INDEX, CACHE_STATUS.WAY) to 'I'.

Table 29-27 CLN_CACHE_CMD Field Description

Field	Bit	Access	Description
OP (continued)	[3:0]	RAZ,W	■ INDEX_CLEAN (0110): if status of line (CACHE_STATUS.INDEX, CACHE_STATUS.WAY) indicates that it is modified w.r.t. memory then write-back the line and update its status: M->E and O->S. If the line is already clean or invalid, do nothing. If the line is pending (status is 'P') then wait until it becomes one of 'M', 'O', 'E' or 'S' and then perform the CLEAN operation. ■ INDEX_CLEAN_INVALIDATE (0111): Perform INDEX_CLEAN, followed by INDEX_INVALIDATE. ■ REGION_INDEX_INVALIDATE (1001): Perform INDEX_INVALIDATE, then increment CACHE_STATUS.INDEX and repeat until CACHE_STATUS.INDEX equals 2 to the power of SCM_ACR.CACHE_SETS. This operation requires CACHE_STATUS.ENABLE = 0 for the complete duration of the operation. The operation is immediately aborted and CACHE_STATUS.ERR is set if CACHE_STATUS.ENABLE=1 when the operation starts. If the cache is enabled before the operation completes, the behavior is undefined. ■ REGION_INDEX_CLEAN (1010): Perform INDEX_CLEAN, then increment CACHE_STATUS.INDEX and repeat until CACHE_STATUS.INDEX equals 2 to the power of SCM_ACR.CACHE_SETS. ■ REGION_INDEX_CLEAN_INVALIDATE (1011): Perform INDEX_CLEAN_INVALIDATE, then increment CACHE_STATUS.INDEX and repeat until CACHE_STATUS.INDEX equals 2 to the power of SCM_ACR.CACHE_SETS. ■ REGION_ADDR_INVALIDATE (1101): invalidate any line in the cache whose block address is in the inclusive range [{CACHE_ADDR_LO1, CACHE_ADDR_LO0,}, {CACHE_ADDR_HI1, CACHE_ADDR_HI0}]. The number of lookups required to execute this operation is never larger than the number of lines in the cache. ■ REGION_ADDR_CLEAN (1110): writeback (if Modified or Owned) any line in the cache whose block address is in the inclusive range [CACHE_ADDR_LO, CACHE_ADDR_HI], then update MOESI state accordingly (i.e. M->E and O->S). The number of lookups required to execute this operation is never larger than the number of lines in the cache. ■ REGION_ADDR_CLEAN_INVALIDATE (1111): writeback (if Modified or Owned) any line in the cache whose block address is in the inclusive range [{CACHE_ADDR_LO1, CACHE_ADDR_LO0,}, {CACHE_ADDR_HI1, CACHE_ADDR_HI0}], then invalidate. A line that is not in M or O state is only invalidated. The number of lookups required to execute this operation is never larger than the number of lines in the cache. All other OP values are reserved and ignored on write. @reset=NOP.

Table 29-27 CLN_CACHE_CMD Field Description

Field	Bit	Access	Description
INCR	[4]	R, W	If 0 then the REGION_INDEX_* commands do not increment CACHE_STATUS.WAY. If 1 then the REGION_INDEX_* commands iterate over all available ways (as indicated by SCM_ACR.CACHE_ASSOC) before incrementing the index and resetting WAY to 0. @reset=0.
RDOMID	[7:5]	R, W	The resource domain used with the quality of service features for the operations triggered by CACHE_CMD.OP. Ignored if QoS is not enabled. @reset=0.
RESERVED	[31:8]	RAZ, IOW	Reserved.

29.4.6 Cache Status and Operation Result Register, CLN_CACHE_STATUS

CLNR Address: 0xD1

Access: RW

The shared cache can be enabled and/or disabled at any point in time without causing malfunctioning of the processor. The result of any individual transaction that is pending during the enable or disable switchover is undefined. It is not defined if any such pending transaction behaves as if the cache is enabled or as if the cache is disabled.

Figure 29-26 CLN_CACHE_STATUS Register

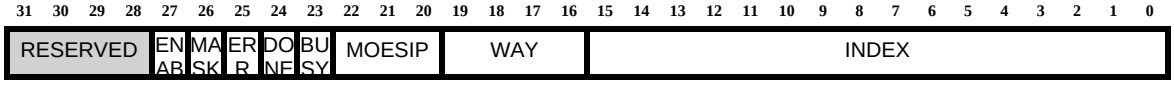


Table 29-28 CLN_CACHE_STATUS Field Description

Field	Bit	Access	Description
INDEX	[15:0]	R, W	Shared cache set index. @reset=0.
WAY	[19:16]	R, W	Shared cache way. @reset=0.
MOESIP	[22:20]	R,IOW	MOESIP status: M (010), O (011), E (100), S (101), I (000), P (001). Other values reserved. @reset=I (000).
BUSY	[23]	R,IOW	'1' while a CLN_CACHE_CMD.OP operation is in progress, 0 otherwise. The DONE field will be set when BUSY makes a 1-to-0 transition. @reset=0.
DONE	[24]	R, W	Cache operation has completed. When MASK is '1' then DONE causes the 'attn' signal to be raised. The 'attn' signal can be cleared by writing a '0' to DONE, or by starting a new region cache operation. Any attempt to write a '1' is ignored. @reset=0.
ERR	[25]	R, W	Cache operation is invalid: an index operation with INDEX, WAY outside the configured cache size, or a region operation with CACHE_ADDR_HI < CACHE_ADDR_LO, or any new CACHE_CMD.OP when BUSY is still '1'. The ERR bit can be cleared by writing a '0', or by starting a new cache operation. Any attempt to write a '1' is ignored. @reset=0.
MASK	[26]	R, W	Interrupt through 'attn' is enabled (1) or disabled (0). @reset=0.
ENABLE	[27]	R, W	If 0 then the shared cache is disabled and bypassed for all transactions. If 1 then the shared cache is enabled. @reset=0.
RESERVED	[31:28]	RAZ, IOW	Reserved.

29.4.7 Cache Writeback Error Status Register, CLN_CACHE_ERR

CLNR Address: 0xD2

Access: RW

Figure 29-27 CLN_CACHE_ERR Register

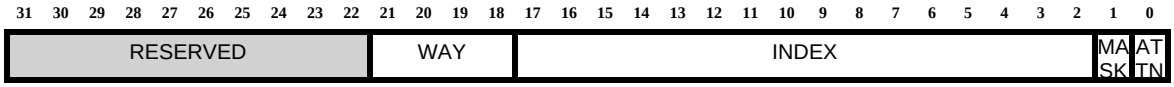


Table 29-29 CLN_CACHE_ERR Field Description

Field	Bit	Access	Description
ATTN_WB	[0]	R, W	If 1 then a writeback operation received a write error response. The address of the faulting line is in CLN_CACHE_ERR_ADDR0 and CLN_CACHE_ERR_ADDR1. In case of multiple faulting writebacks, only the first error is captured. Once the ATTN_WB bit is cleared by writing a 0, it is ready to capture new faulting writebacks. @reset=0.
MASK_WB	[1]	R, W	If 1 then ATTN_WB contributes to the outgoing 'attn' signal. @reset=0.
INDEX	[17:2]	R,IOW	The shared cache set index from where the faulting write originates. @reset=0.
WAY	[21:18]	R,IOW	The shared cache way from where the faulting write originates. @reset=0.
RESERVED	[31:22]	RAZ, IOW	Reserved.

29.4.8 Faulting Address Bits [31:0] Register, CLN_CACHE_ERR_ADDR0

CLNR Address: 0xD3

Access: R

Figure 29-28 CLN_CACHE_ERR_ADDR0

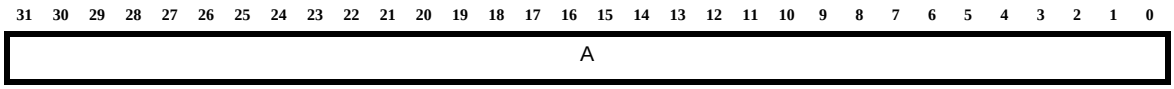


Table 29-30 CLN_CACHE_ERR_ADDR0 Field Description

Field	Bit	Access	Description
A	[31:0]	R, IOW	Bits [31:0] of the faulting writeback address.

29.4.9 Faulting Address Bits [51:32] Register, CLN_CACHE_ERR_ADDR1

CLNR Address: 0xD4

Access: R

Figure 29-29 CLN_CACHE_ERR_ADR1 Register

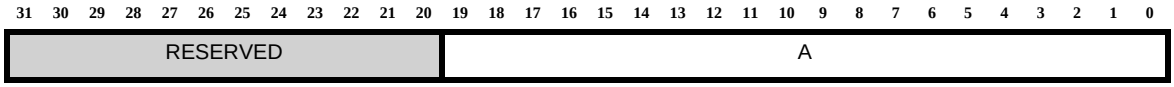


Table 29-31 CLN_CACHE_ ERR _ADR1 Description

Field	Bit	Access	Description
A	[19:0]	R, IOW	Bits [51:32] of the faulting writeback address.
RESERVED	[31:20]	RAZ, IOW	Reserved.

29.5 Programming Master Port Address Apertures

Sections 29.5.1 and 29.5.2 defines the base address and size of a NoC master port aperture. Master port aperture sizes are always multiples of 1 MB, and their base address is always aligned on a 1 MB boundary. There can be up to 8 apertures associated with a single NoC master port, as specified by `CFG_MST_NOC[i]`.num_aperture.

The CLN can be configured with `CFG_MST_RST_APERTURE`=1 which will provision the CLN with a set of pins {`mst_rst_aperture_regid`[11:0], `mst_rst_aperture_addr`[31:0], `mst_rst_aperture_size`[31:0]}. The `mst_rst_aperture_regid` specifies the CLN register id of a `CLN_MST_NOC_i_ADDR` register, or a `CLN_MST_CCM_i_k_ADDR` register. The other pins provide the base address and size that will be loaded in the specified CLN address and size registers at reset. The size register id is assumed to be the base address register id plus 1.

29.5.1 NoC Master #i Aperture #j Lowest Address Register, CLN_MST_NOC_*i*_*j*_ADDR

CLNR Address: See Table 29-3.

Access: RW

Figure 29-30 CLN_MST_NOC_*i*_*j*_ADDR Register

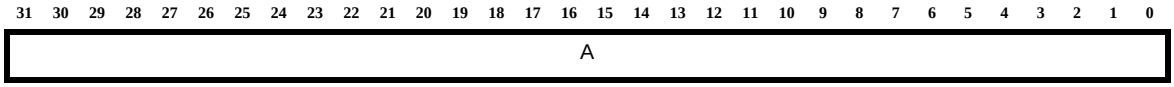


Table 29-32 CLN_MST_NOC_*i*_*j*_ADDR Field Description

Field	Bit	Access	Description
A	[31:0]	R, W	Bits [51:20] of the aperture base address. Bits [19:0] are assumed all 0's. @reset=0, or copied from the mst_rst_aperture_addr pins if so configured with CFG_MST_RST_APERTURE.

29.5.2 Size NoC Master #i Aperture #j Size Register, CLN_MST_NOC_*i*_*j*_SIZE

CLNR Address: See Table 29-3.

Access: RW

Figure 29-31 CLN_MST_NOC_*i*_*j*_SIZE Register

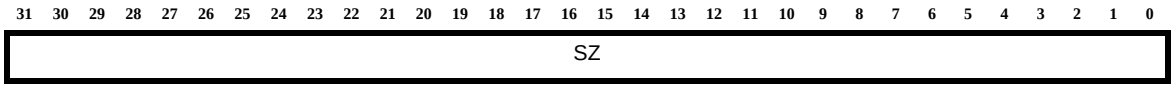


Table 29-33 CLN_MST_NOC_*i*_*j*_SIZE Field Description

Field	Bit	Access	Description
SZ	[31:0]	R, W	Size of the aperture in multiples of 1 MB. @reset=0, or copied from the mst_rst_aperture_size pins if so configured with CFG_MST_RST_APERTURE.

29.5.3 CCM Master #i Aperture #k Lowest Address Register, CLN_MST_CCM_*i*_*k*_ADDR

CLNR Address: 0x300+8i+2k
i-[0-1F]
k-[0-3]

Access: RW

Figure 29-32 CLN_MST_CCM_*i*_*k*_ADDR Register

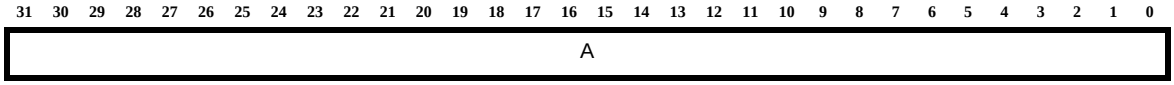


Table 29-34 CLN_MST_CCM_*i*_*k*_ADDR Field Description

Field	Bit	Access	Description
A	[31:0]	R, W	Bits [51:20] of aperture *k* base address associated with the CCM *i* master port. Bits [19:0] are assumed all 0's. The base address must be a multiple of the aperture size as specified in CLN_MST_CCM_*i*_*k*_size. @reset=0, or copied from rst_aperture pins if so configured with CFG_MST_RST_APERTURE.

29.5.4 CCM Master #i Aperture #k Size Register, CLN_MST_CCM_*i*_*k*_SIZE

CLNR Address: 0x301+8i+2k
i-[0-1F]
k-[0-3]

Access: RW

Figure 29-33 CLN_MST_CCM_*i*_*k*_SIZE Register

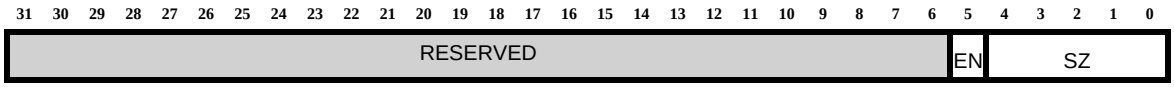


Table 29-35 CLN_MST_CCM_*i*_*k*_SIZE Field Description

Field	Bit	Access	Description
SZ	[4:0]	R, W	Log2 of the size (in 64-byte blocks) of aperture k of CCM master port i. The aperture size is therefore always a power of 2. The reset value of this field is zero unless the `CFG_MST_RST_APERTURE_REGID` pins at reset time are tied to the address of the corresponding `CLN_MST_CCM_i_k_ADDR` register. In this case, the reset value is copied from the `CFG_MST_RST_APERTURE_SIZE` input pins.
EN	[5]	R,W	Enable (1) or disable (0) aperture *k* of CCM master port *i*.
RESERVED	[31:6]	RAZ, IOW	Reserved.

There can be up to two peripheral master ports, associated with two peripheral apertures. "PERj Peripheral Aperture Lowest Address Register, CLN_PERj_BASE" and "PERj Peripheral Aperture Size Register, CLN_PERj_SIZE" describes the base address and size of each peripheral aperture.

29.5.5 PERj Peripheral Aperture Lowest Address Register, CLN_PER*j*_BASE

CLNR Address: 0xA80 and 0xA82

Access: RW

Figure 29-34 CLN_PER*j*_BASE Register

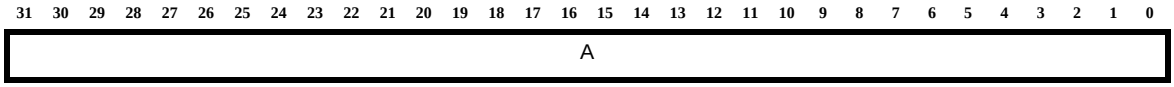


Table 29-36 CLN_PER*j*_BASE Field Description

Field	Bit	Access	Description
A	[31:0]	R, W	Bits [51:20] of the aperture base address. Bits [19:0] are assumed all 0's. @reset=0, or copied from the mst_rst_aperture_addr pins if so configured with CFG_MST_RST_APERTURE.

29.5.6 PERj Peripheral Aperture Size Register, CLN_PER*j*_SIZE

CLNR Address: 0xA81 and 0xA83

Access: RW

Figure 29-35 CLN_PER*j*_SIZE Register

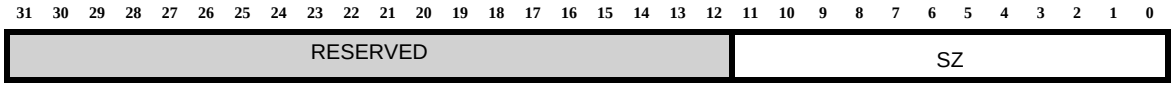


Table 29-37 CLN_PER*j*_SIZE Field Description

Field	Bit	Access	Description
SZ	[11:0]	R, W	Size of the aperture in multiples of 1 MB. @reset=0, or copied from the mst_rst_aperture_size pins if so configured with CFG_MST_RST_APERTURE.
RESERVED	[31:12]	RAZ, IOW	Reserved.

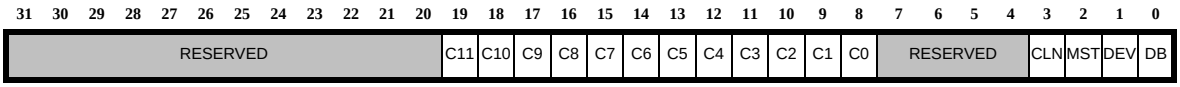
29.5.7 Global Clock Disable Register, GLOBAL_CLK_GATE_DIS

Address: 0x9A9

Access: RW

Reset: 000F_FF0F

Figure 29-36 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 29-38 GLOBAL_CLK_GATE_DIS Register

Field	Bit	Description
DB	[0]	Disable L1 clock gating of SCM data banks. This field is present even if SCM is not configured. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
DEV	[1]	Disable L1 clock gating of all device target ports.@reset=1. ■ `0x0`: enable ■ `0x1`: disable

Table 29-38 GLOBAL_CLK_GATE_DIS Register

Field	Bit	Description
MST	[2]	Disable L1 clock gating of the CCM, NOC, and peripheral initiator ports. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
CLN	[3]	Disable L1 clock gating of the cluster network internal components (such as shared memory cache, and so on). @reset=1. ■ `0x0`: enable ■ `0x1`: disable
RESERVED	[7:4]	Reserved. Read as zero and ignored on writes.
C0	[8]	Enable dynamic clock gating at level 1 for ARC core 0. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C1	[9]	Enable dynamic clock gating at level 1 for ARC core 1. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C2	[10]	Enable dynamic clock gating at level 1 for ARC core 2. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C3	[11]	Enable dynamic clock gating at level 1 for ARC core 3. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C4	[12]	Enable dynamic clock gating at level 1 for ARC core 4. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C5	[13]	Enable dynamic clock gating at level 1 for ARC core 5. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C6	[14]	Enable dynamic clock gating at level 1 for ARC core 6. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C7	[15]	Enable dynamic clock gating at level 1 for ARC core 7. @reset=1. ■ `0x0`: enable ■ `0x1`: disable

Table 29-38 GLOBAL_CLK_GATE_DIS Register

Field	Bit	Description
C8	[16]	Enable dynamic clock gating at level 1 for ARC core 8. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C9	[17]	Enable dynamic clock gating at level 1 for ARC core 9. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C10	[18]	Enable dynamic clock gating at level 1 for ARC core 10. @reset=1. ■ `0x0`: enable ■ `0x1`: disable
C11	[19]	Enable dynamic clock gating at level 1 for ARC core 11. @reset=1. ■ `0x0`: enable ■ `0x1`: disable

29.5.8 ARC Cores Power Status Register, CLN_PWR_STATUS_0

Address: 0x17C

Access: R

The `CLN_PWR_STATUS_0` register indicates power information of all ARC cores in cluster. If `power_domains` option is `false`, this register is reserved and attempt to access this register result in a bus error. This register is available in the cluster network auxiliary space.

Figure 29-37 CLN_PWR_STATUS_0 Register

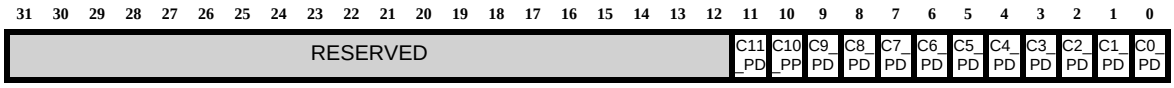


Table 29-39 CLN_PWR_STATUS_0 Field Description

Field	Bit	Access	Description
C*n*_PD	[*n*]	R,IOW	Power status of core *n*. ■ `0x0`: core *n* is powered-on currently ■ `0x1`: core *n* is powered-down currently If the core *n* is not present in the system. Reads to this bit return 0.
RESERVED	[31:12]	RAZ, IOW	Reserved.

Note The indexes of Core<x> and accelerator<y> are determined by the building order in ARChitect2.

29.5.9 Initiator Port Power Status Register, CLN_PWR_STATUS_1

Address: 0x17D

Access: R

The `CLN_PWR_STATUS_1` register indicates power information of slave ports connected to devices. If `power_domains` option is `false`, this register is reserved and attempt to access this register result in a bus error.

Figure 29-38 CLN_PWR_STATUS_1 Register (4 Accelerators Example)

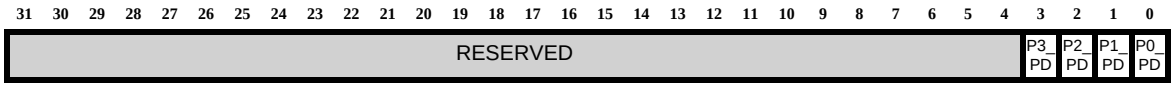


Table 29-40 CLN_PWR_STATUS_1 Field Description

Field	Bit	Access	Description
P*n*_PD	[*n*] *n* belongs to [31:0]	R,IOW	Power status of initiator port *n*. ■ `0x0`: initiator port *n* is powered-on currently ■ `0x1`: initiator port *n* is powered-down currently If the initiator port *n* is not present in the system. Reads to this bit return 0.

Note The index of port<x> is determined by the building order in ARChitect2

29.5.10 Device Target Port Power Status Register, CLN_PWR_STATUS_2

Address: 0x17E

Access: R

The `CLN_PWR_STATUS_2` register indicates power information of device target ports. If `power_domains` option is `false`, this register is reserved and attempt to access this register result in a bus error.

Figure 29-39 CLN_PWR_STATUS_2 Register (4 Accelerators Example)

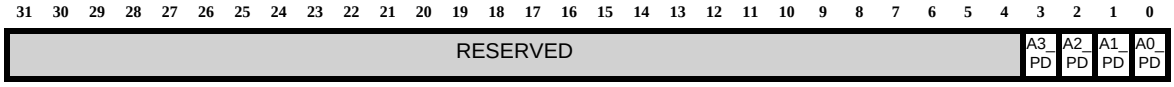


Table 29-41 CLN_PWR_STATUS_2 Field Description

Field	Bit	Access	Description
A*n*_PD	[*n*] *n* belongs to [31:0]	R,IOW	Power status of accelerator *n*. ■ `0x0`: accelerator *n* is powered-on currently ■ `0x1`: accelerator *n* is powered-down currently If the accelerator *n* is not present in the system. Reads to this bit return 0.

29.5.11 ARC Trace Power Status Register, CLN_RTT_PDM_STATUS

Address: 0x17F

Access: R

This register indicates indicates power mode of all trace clients in the cluster. The value is the snapshot of `RTT_PDM_PSTAT[CPM]` field of each client. If `power_domains` option is `false`, this register is reserved and attempt to access this register result in a bus error.

Figure 29-40 CLN_RTT_PDM_STATUS Register

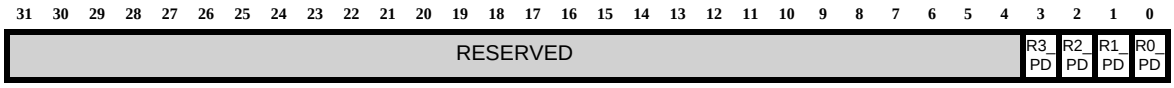


Table 29-42 CLN_RTT_PDM_STATUS Field Description

Field	Bit	Access	Description
R*n*_PD	[*n*] *n* belongs to [31:0]	R,IOW	Power status of ARC Trace client *n*. ■ `0x0`: ARC Trace client *n* is powered-on currently ■ `0x1`: ARC Trace client *n* is powered-down currently If the ARC Trace client *n* is not present in the system. Reads to this bit return 0.

29.5.12 Programming ARC Trace Transport Parameters

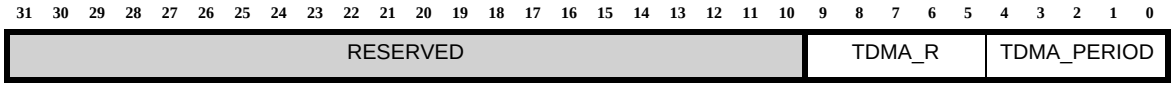
When an ARC core provides a stream of trace messages then register CLN_SLV_i_ARCT can be programmed with the TDMA parameters that guarantee freedom of interference between trace messages and regular memory traffic.

29.5.13 ARC Trace Transport Register, CLN_SLV_i_ARCT

CLNR Address: to 0xAE0 to 0xAEB

Access: RW

Figure 29-41 CLN_SLV_i_ARCT Register



When an ARC core provides a stream of trace messages then register CLN_SLV_i_ARCT can be programmed with the TDMA parameters that guarantee freedom of interference between trace messages and regular memory traffic.

Table 29-43 CLN_SLV_i_ARCT Field Description

Field	Bit	Access	Description
TDMA_PERIOD	[4:0]	R, W	One less than the length of a TDMA interval expressed as number of cycles of the clock associated with the SLV i port. @reset=31.
TDMA_RESERVATION (TDMA_R)	[9:5]	R,W	The number of cycles within each (1 + TDMA_PERIOD) that is reserved for trace messages. These reserved cycles cannot be used by regular memory traffic. Trace messages can use additional cycles when these are not claimed by regular traffic. If TDMA_RESERVATION is zero, or if TDMA_RESERVATION > TDMA_PERIOD, then TDMA is disabled. @reset=0.

29.5.14 Programming QoS Parameters

The QoS features and the following CLNR registers to control them are only available if the configuration parameter CFG_QOS is non-zero. QoS settings can be changed at any time without affecting the correct operation of the cluster.

Quality of Service is only available after it has been enabled through the `CLN_QOS_CTL` register.

29.5.15 QOS Control Common to all Resource Domains Register, CLN_QOS_CTL

CLNR Address: 0x180

Access: RW

Figure 29-42 CLN_QOS_CTL Register

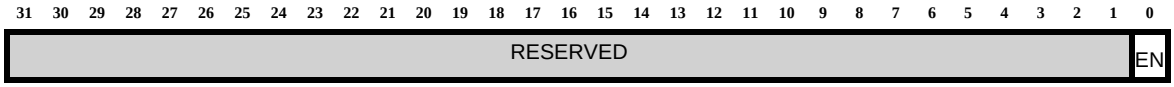


Table 29-44 CLN_QOS_CTL Field Description

Field	Bit	Access	Description
EN	[0]	R, W	Enable (1) or disable (0) all QoS features. @reset=0.
RESERVED	[31:1]	RAZ, IOW	Reserved.

It is possible for more than one resource domain to reserve the same channel. In that case there is no mutual priority for that transport channel, and if anyone of the domains reserving a channel makes it transferable by setting the associated XFER bit then it becomes transferable also for the other domains with the same channel reservation.

29.5.15.1 QoS Cache Footprint Control for Resource Domain #i Register, CLN_QOS_RDOMi_FOOTPRINT

CLNR Address: 0x18A+8i
i ~ [0-7]

Access: RW

Figure 29-43 CLN_QOS_RDOMi_FOOTPRINT Register

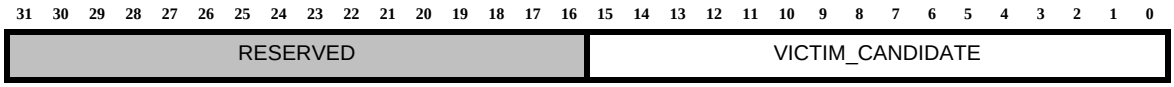


Table 29-45 CLN_QOS_RDOMi_FOOTPRINT Field Description

Field	Bit	Access	Description
VICTIM_CANDIDATE	[15:0]	R, W	A '1' in position *x* indicates that way *x* of the shared cache is included in the set of permissible victims. A '0' indicates that way *x* cannot be used to select a victim whenever a refill occurs on behalf of resource domain *i*. On write this saturates to the reset value. @reset=2 ^ CFG_SCM_CACHE_ASSOC – 1.
RESERVED	[31:16]	RAZ, IOW	Reserved.

29.5.15.2 QoS Cache Slave Port #i Register, CLN_SLV_*i*_QOS

CLNR Address: 0xAC0 to 0xADF

Access: RW

Figure 29-44 CLN_SLV_*i*_QOS Register

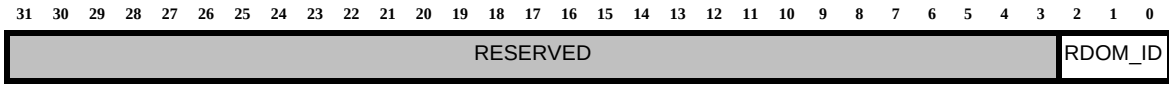


Table 29-46 CLN_SLV_*i*_QOS Field Description

Field	Bit	Access	Description
RDOM_ID	[2:0]	R, W	The QoS resource domain associated with all transactions originating from slave port #i. An OS with dynamic thread scheduling and quality of service rules typically updates this register as part of the thread scheduling process. @reset=0.
RESERVED	[31:3]	RAZ, IOW	Reserved

29.6 Error Conditions and Job Completion

Error conditions are normally signaled with an appropriate value on the read response or write response channel, as specified by IBPv3. There are cases where the cluster initiates a read or write transaction without an associated transaction on any of the slave ports that could handle the error response. These cases include shared cache prefetches and shared cache writeback operations.

The cluster also supports posted writes (which are 'buffered' writes according to their `cmd_cache[]` attributes). These writes receive an early write response `wr_done` which completes the transaction at the slave port side. When later the write data is forwarded to the final destination, the target can still respond with an error condition (`wr_err`). In this case the cluster records the address and transaction details in the registers `CLN_WR_ERR, CLN_WR_ERR_ADDR0` and `CLN_WR_ERR_ADDR1`.

29.6.1 Posted Write Error Status Register, CLN_WR_ERR

CLNR Address: 0x680

Access: RW

Figure 29-45 CLN_WR_ERR Register

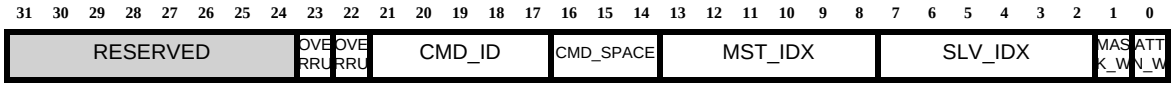


Table 29-47 CLN_WR_ERR Field Description

Field	Bit	Access	Description
ATTN_WR	[0]	R, W	If 1 then a posted write, or a shared cache writeback operation, or the write-portion of an AtomicSwap, received a write error response from the final target. The address of the faulting transaction is in CLN_WR_ERR_ADDR0 and CLN_CACHE_WR_ADDR1. In case of multiple faulting writes, only the first error is captured. Once the ATTN_WR bit is cleared by writing a 0, it is ready to capture new faulting posted writes. @reset=0.
MASK_WR	[1]	R, W	If 1 then ATTN_WR contributes to the outgoing 'attn' signal. @reset=0.
SLV_IDX	[7:2]	R, IOW	The slave interface from where the offending write was initiated. Set to 'd63 if slave port is unknown, e.g. with a writeback error. @reset=0.
MST_IDX	[13:8]	R, IOW	The master interface where the write error response was received. @reset=0.
CMD_SPACE	[16:14]	R, IOW	The cmd_space of the offending posted write. @reset=0.
CMD_ID	[21:17]	R, IOW	The cmd_id of the offending posted write. @reset=0.
OVERRUN_SAME	[22]	R, IOW	Set to '1' if additional write errors occur on the same slave port (captured by SLV_IDX) while ATTN_WR is still pending. @reset=0.
OVERRUN_OTHER	[23]	R, IOW	Set to '1' if write errors occur on other slave ports (i.e. other than indicated by the SLV_IDX field) while ATTN_WR is still pending. @reset=0.
RESERVED	[31:24]	RAZ, IOW	Reserved.

29.6.2 Posted Write Error Address Bits [31:0] Register, CLN_WR_ERR_ADDR0

CLNR Address: 0x681

Access: R

Figure 29-46 CLN_WR_ERR_ADDR0 Register

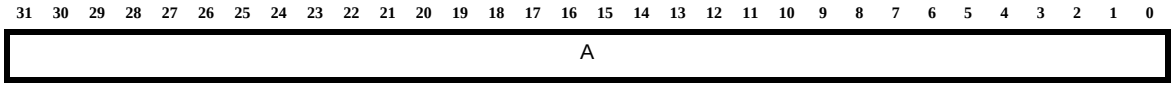


Table 29-48 CLN_WR_ERR_ADDR0 Field Description

Field	Bit	Access	Description
A	[31:0]	R,IOW	Bits [31:0] of the faulting posted write address.

29.6.3 Posted Write Error Address Bits [51:32] Register, CLN_WR_ERR_ADDR1

CLNR Address: 0x682

Access: R

Figure 29-47 CLN_WR_ERR_ADDR1 Register

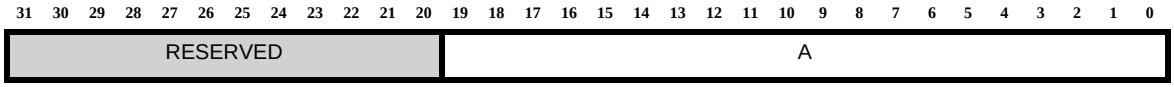


Table 29-49 CLN_WR_ERR_ADDR1 Field Description

Field	Bit	Access	Description
A	[19:0]	R,IOW	Bits [31:0] of the faulting posted write address.
RESERVED	[31:20]	RAZ, IOW	Reserved.

The cluster includes an outgoing signal 'attn' that can be wired to an interrupt controller or any other means to capture the attention of an operating system.

In addition to signaling errors, the 'attn' signal can also be used to signal the completion of a cache region flush or invalidate operation. The 'attn' signal is also used by the Cluster Performance Counters to raise an interrupt, as described in "Cluster Performance Counter Registers" on page 1222. If an interrupt controller is configured in the ARC cores then the 'attn' signal is internally wired to irq23 of each ARC core.

29.6.4 'attn' Interrupt Signal Status Register, CLN_ATTN

CLNR Address: 0x690

Access: R

Figure 29-48 CLN_ATTN Register

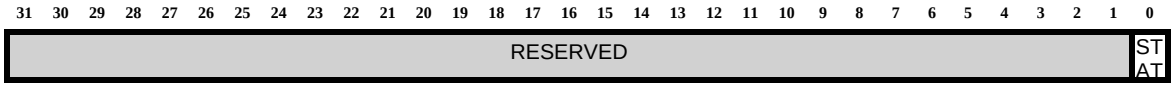


Table 29-50 CLN_ATTN Field Description

Field	Bit	Access	Description
STATUS	[0]	R,IOW	Reflects the current state of the outgoing 'attn' wire. If 1 this means that any other CLN register has its ATTN bit raised. The STATUS bit can be cleared only by clearing the underlying ATTN condition. If multiple such conditions are raised then all will have to be cleared before STATUS returns to 0.
RESERVED	[31:1]	RAZ, IOW	Reserved.

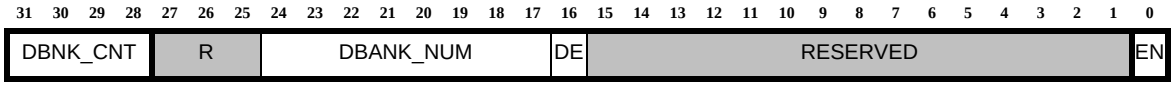
29.7 Error Protection Registers

Each memory block in the cluster network SCM, SCU, and CDMA has an ECC control register.

29.7.1 Data Bank ECC Control Register, CLN_DBANK_ECC_CTRL

CLNR Address:	0x960
Access:	RW
Reset:	0x0000_0000

Figure 29-49 CLN_DBANK_ECC_CTRL Register



This register allows you to enable or disable ECC protection on the shared cluster data memory. This register is updated when a double-bit error occurs in the memory and also the number of corrected single-bit errors.

If `cfg_scm_ecc` is configured, the data bank memory has an ECC code for each memory entry which is 128 bits.

The memory layout is as in the following diagram. The ECC code is located in the MSBs of each entry.

Both ECC encode and ECC decode are pipelined. So it could only introduce dbank access latency by 1 cycle per burst and does not affect the memory bandwidth. If a single or burst write transaction has beats that are not full, the dbank memory controller must read that entry from memory before performing the intended write. At most one read operation could be trigger per beat.

If a single-bit error is detected from this memory, the error is corrected before returning the initiator and the single-bit error counter field, ECC_DBNK_SBE_CNT, is incremented.

If a two-bit error is detected from this memory, an error response is returned to its initiator along with the return data. The uncorrectable error bit and error bank number fields in the CLN_DBNK_ECC_CTRL register are updated if those fields are still clear.

If ECC is configured, there is a dedicated response channel between the CLN core and the data bank memory. It is used to carry ECC error status to the AUX register module and hold write transactions that are not posted.

Table 29-51 CLN_DBANK_ECC_CTRL Field Description

Field	Bit	Description
EN	[0]	Control bit for protection on the shared cluster memory data bank; A value of 1 enables protection, and 0 disables protection. On reset, the control bits are set to 0; that is ECC protection is not enabled.
RESERVED	[15:1]	Read as zero and ignored on writes.
ECC_DBNK_ERR (DE)	[16]	Two-bit error has been detected. Write 0 to clear this bit.
DBANK_NUM	[24:17]	Indicates the memory bank that encountered an uncorrectable error.
RESERVED	[27:25]	Read as zero and ignored on writes.
ECC_DBNK_SBE_CNT(DBNK_CNT)	[31:28]	Indicates the number of corrected single-bit errors on the memory. If the error count reaches the maximum value, the counter retains the maximum value and subsequent single-bit errors are ignored. Write 0 to clear the count value.

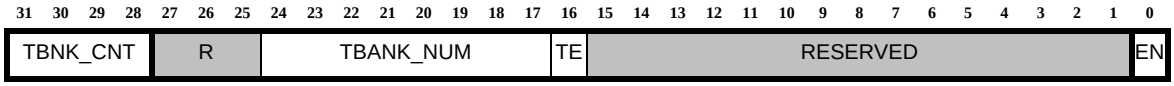
29.7.2 Tag Bank ECC Control Register, CLN_TBNK_ECC_CTRL

CLNR Address: 0x961

Access: RW

Reset: 0x0000_0000

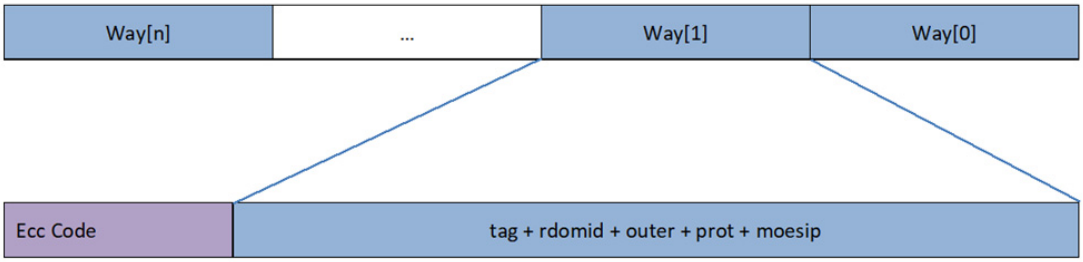
Figure 29-50 CLN_TBNK_TAG_CTRL Register



This register allows you to enable or disable ECC protection on the shared cluster tag memory. This register is updated when a double-bit error occurs in the memory and also the number of corrected single-bit errors.

If cfg_scm_ecc is configured, the share cache tag bank memory can be configured with ECC protection function.

The share cache tag bank memory has multiple ways. Each way has its own ECC code. The following diagram shows the layout of ECC and tag in the tag bank memory in one of the banks.



Each tag update operates on one tag way and it writes to all bits of that entry. No read-modify-write are needed for this memory.

For tag lookup, all ways are read at the same cycle and all ECC codes are checked in parallel with hit/miss determination. Any error found in a lookup causes the cluster network to start to process error reporting and all hit/miss results from that access is dropped.

If ECC is configured, the dbnk command generated by tag lookup has a ecc_error bit. That bit is used to indicate that the lookup result is not trustable and the consumer modules must take action for that error.

Table 29-52 CLN_TBNK_ECC_CTRL Field Description

Field	Bit	Description
EN	[0]	Control bit for protection on the shared cluster tag memory bank; A value of 1 enables protection, and 0 disables protection. On reset, the control bits are set to 0; that is ECC protection is not enabled.
RESERVED	[15:1]	Read as zero and ignored on writes.
ECC_TBNK_ERR (TE)	[16]	Two-bit error has been detected. Write 0 to clear this bit.
TBANK_NUM	[24:17]	Indicates the memory bank that encountered an uncorrectable error.
RESERVED	[27:25]	Read as zero and ignored on writes.
ECC_TBNK_SBE_CNT(TBNK_CNT)	[31:28]	Indicates the number of corrected single-bit errors on the memory. If the error count reaches the maximum value, the counter retains the maximum value and subsequent single-bit errors are ignored. Write 0 to clear the count value.

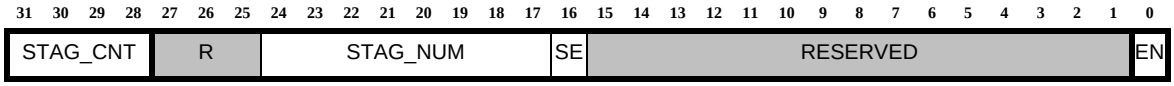
29.7.3 SCU Shadow Tag Memory ECC Control Register, CLN_STAG_ECC_CTRL

CLNR Address: 0x962

Access: RW

Reset: 0x0000_0000

Figure 29-51 CLN_STAG_ECC_CTRL Register



This register allows you to enable or disable ECC protection on the shared coherency unit shadow memory. This register is updated when a double-bit error occurs in the memory and also the number of corrected single-bit errors.

If `cfg_scu_ecc` is configured, the shadow tag memory can be configured with ECC memory protection. The SCU shadow tag memory has multiple ways. Each way has its own ECC code. The following diagram shows the layout of ECC and tag in the shadow tag memory in one of the banks.

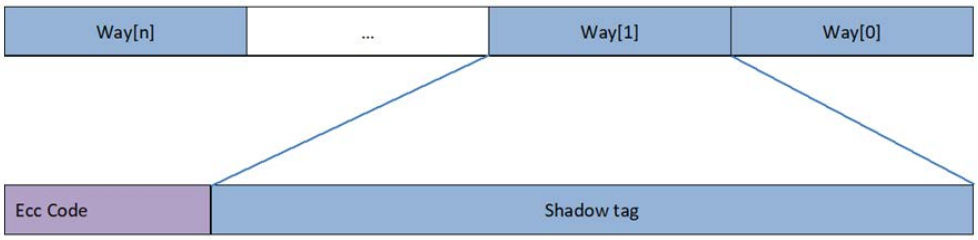


Table 29-53 CLN_STAG_ECC_CTRL Field Description

Field	Bit	Description
EN	[0]	Control bit for protection on the shared coherency unit shadow tag memory bank; A value of 1 enables protection, and 0 disables protection. On reset, the control bits are set to 0; that is ECC protection is not enabled.
RESERVED	[15:1]	Read as zero and ignored on writes.
ECC_STAGE_ERR (SE)	[16]	Two-bit error has been detected. Write 0 to clear this bit.
STAG_NUM	[24:17]	Indicates the memory bank that encountered an uncorrectable error.

Table 29-53 CLN_STAG_ECC_CTRL Field Description

Field	Bit	Description
RESERVED	[27:25]	Read as zero and ignored on writes.
ECC_STAG_SBE_CNT(TBNK_CNT)	[31:28]	Indicates the number of corrected single-bit errors on the memory. If the error count reaches the maximum value, the counter retains the maximum value and subsequent single-bit errors are ignored. Write 0 to clear the count value.

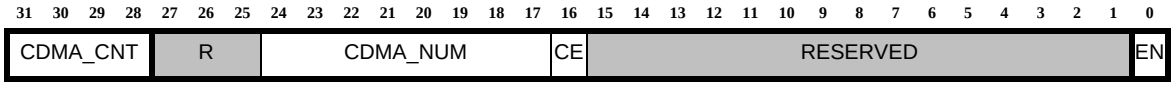
29.7.4 DMA Descriptor Memory ECC Control Register, CLN_CDMA_ECC_CTRL

CLNR Address: 0x963

Access: RW

Reset: 0x0000_0000

Figure 29-52 CLN_CDMA_ECC_CTRL Register



This register allows you to enable or disable ECC protection on the cluster DMA descriptor memory. This register is updated when a double-bit error occurs in the memory and also the number of corrected single-bit errors.

If `dma_srv_desc_ram_ecc` is configured, the cluster DMA (CDMA) descriptor memory can be configured with ECC protection. The ECC code is appended in the MSBs of each CDMA descriptor. The CDMA performs ECC code generation and error detection on the fly. If one-bit error is detected, that error is corrected and the command is processed. If two-bit error is detected, the command is not processed.

Table 29-54 CLN_CDMA_ECC_CTRL Field Description

Field	Bit	Description
EN	[0]	Control bit for protection on the cluster DMA descriptor memory bank; A value of 1 enables protection, and 0 disables protection. On reset, the control bits are set to 0; that is ECC protection is not enabled.
RESERVED	[15:1]	Read as zero and ignored on writes.
ECC_CDMA_ERR (CE)	[16]	Two-bit error has been detected. Write 0 to clear this bit.
CDMA_NUM	[24:17]	Indicates the memory bank that encountered an uncorrectable error.
RESERVED	[27:25]	Read as zero and ignored on writes.
ECC_CDMA_SBE_CNT(TBNK_CNT)	[31:28]	Indicates the number of corrected single-bit errors on the memory. If the error count reaches the maximum value, the counter retains the maximum value and subsequent single-bit errors are ignored. Write 0 to clear the count value.

29.8 Cluster Auxiliary and Build Configuration Registers

Table 29-55 Cluster Registers

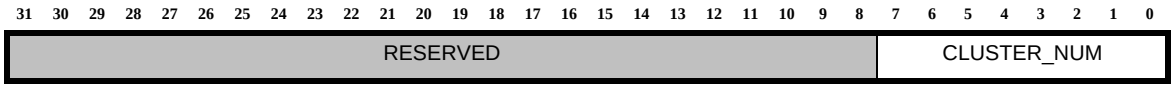
Address	Auxiliary Register Name	Description
0x298	CLUSTER_ID	Cluster ID register

29.8.1 CLUSTER ID Register, CLUSTER_ID

Address: 0x298

Access: r

Figure 29-53 CLUSTER_ID Register



This register contains an 8 bit cluster identification number that is assigned using the input signal: `clusternum[7:0]`.

Table 29-56 CLUSTER_ID Register

Field	Bit	Description
CLUSTER_NUM	[7:0]	Contains the cluster identification number.
RESERVED	[31:8]	Reserved. Read as zero ignored on write.

In a multi-core configuration of the cluster, every connected master is allocated its own physical copy of `CLNR_ADDR`, even though each master is using the same AUX address to access its copy of `CLNR_ADDR`. This avoids race conditions if more than one master tries to access the shared CLNR space at the same time. The `CLNR_ADDR` can be read which is useful in a save-restore sequence, for example, when a debugger is preempting and later resuming software that may be accessing the CLN register space.

29.9 Cluster Performance Counter Registers

Table 29-57 Cluster Performance Counter Registers

CLN Address	Name	R/W	Description	Comments
0xC03	CPCT_CC_NUM Register	RW	Countable condition number register	
0xC04 to 0xC07	Countable Conditions Name Registers, CPCT_CC_NAMEi	R	Countable condition names	
0xC08	Control Register: CPCT_CONTROL	RW	Control register	-
0xC09	Interrupt Control Register, CPCT_INT_CTRL	RW	Counter Interrupt Enable	(Optional)

Table 29-57 Cluster Performance Counter Registers

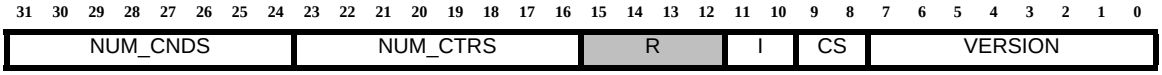
CLN Address	Name	R/W	Description	Comments
0xC0A	Interrupt Active Register, CPCT_INT_ACT	RW	Counter Interrupt Active	(Optional)
0xD00	Configuration Register: CPCT_<N>CONFIG	RW	Counter configuration and control	One per counter
0xD02	Count-Value Registers: CPCT_COUNTL	RW	Lower order current-count value	One per Counter
0xD03	Count-Value Registers: CPCT_COUNTH	RW	Higher order current-count value	One per Counter
0xD04	Snapshot-Value Registers, CPCT_<N>_SNAPL	RW	Lower order snapshot value	One per Counter
0xD06	Interrupt Trigger Value Registers, CPCT_INT_CNTL	RW	Interrupt Trigger Count Low	One per Counter (optional)
0xD07	Interrupt Trigger Value Registers, CPCT_INT_CNTH	RW	Interrupt Trigger Count High	One per Counter (optional)

29.9.1 Cluster Performance Counter Build Configuration Register, CPCT_BUILD

Address: 0xC00

Access: R

Figure 29-54 CPCT_BUILD Register



This register is indicates the presence of the cluster performance counters and their configuration.

Table 29-58 CPCT_BUILD Register Field Description

Field	Range	Description
VERSION	[7:0]	Version number: ■ 0x0: Not present ■ 0x1: Cluster performance counters are present
CS	[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	[11:10]	Indicates the ability of the PCT logic to generate an interrupt on counter overflow. ■ 0x0: interrupt capability excluded ■ 0x1: interrupt capability included
NUM_CTRS	[23:16]	Number of performance counters. Possible values are 0,2,4,8,16,32
NUM_CNDS	[31:24]	Number of countable conditions present in Cluster PCT module

29.9.1.1 CPCT_CC_NUM Register

Address: 0xC03

Access: RW

Figure 29-55 CPCT_CC_NUM

This register is used to set a countable condition number so that the countable condition's name corresponding to that number is loaded into the registers from CPCT_CC_NAME0 to CPCT_CC_NAME3 by hardware.

Table 29-59 CPCT_CC_NUM Register Field Description

Field	Range	Description
CC_NUM	[14:0]	■ Countable condition number corresponding to the ASCII string name viewable in the CPCT_CC_NAME0-3 registers.
RESERVED	[31:15]	Write all zeros to this field.

29.9.1.2 Countable Conditions Name Registers, CPCT_CC_NAMEi

Address: 0xC04 to 0xC07

Access: R

Figure 29-56 CPCT_CC_NAME0 Register

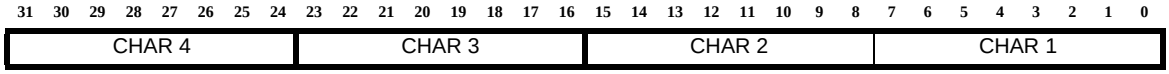


Figure 29-57 CPCT_CC_NAME1 Register

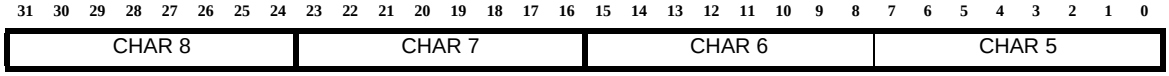


Figure 29-58 CPCT_CC_NAME2 Register

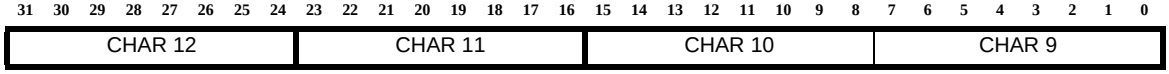
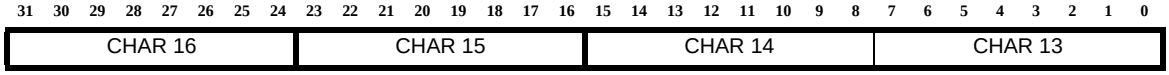


Figure 29-59 CPCT_CC_NAME3 Register



These register provides 1 to N characters of countable condition's name whose index was written into the CPCT_CC_INDEX register, where N is {8 or 12 or 16}. All the fields of these registers are read only accessible.

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 'N' 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 CPCT_CC_INDEX, all zeros (null string) is returned.

CPCT_CC_NAME2 register will present in hardware , if build configuration parameter "-pct_cc_name_size" value is greater than 8.

CPCT_CC_NAME3 register will present in hardaware, if build configuration parameter "-cpct_cc_name_size" value is greater than 12.

29.9.2 Index-Select Register: CPCT_INDEX

Figure 29-60 CPCT_Index

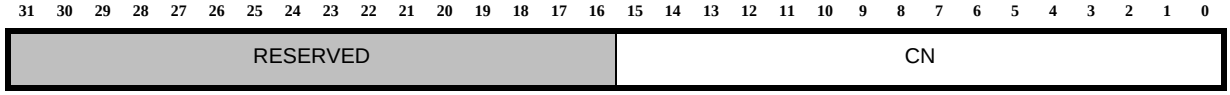


Table 29-60 CPCT_INDEX Register field description

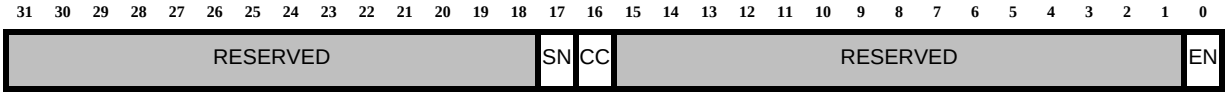
Field	Range	Description
CN	[15:0]	Counter number. Selects the counters to be accessed through count, snapshot and configuration registers. Counter operation is not affected by index selection.
Reserved	[31:16]	IOW/RAZ

29.9.3 Control Register: CPCT_CONTROL

Address: 0xC08

Access: RW

Figure 29-61 CPCT_CONTROL



This register provides single control on all consecutive counters starting from Counter 0 having their LCE bits cleared. Effectively this register could be used to control all counters if all of their LCE bits are cleared. Note that the assertion or change of any control bit could take effect 0 to few cycles after the register is written depending on the implementation.

Table 29-61 CPCT_CONTROL Register Field Descrciption

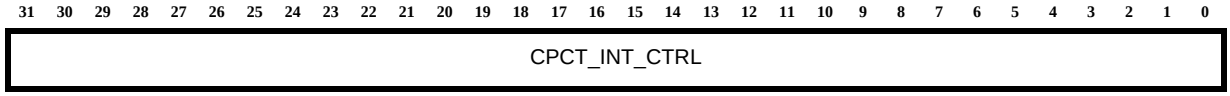
Field	Range	Description
EN	[0]	Enable or disable all consecutive counters starting from Counter 0 having their LCE bits cleared; effectively this bit could be used to enable or disable all counters if all of their LCE bits are cleared. ■ 0x0 – Disable Counting (Global Disable) ■ 0x1 – Enable Counting (Global Enable)
Reserved	[15,1]	Reserved. Read as zero and ignore on writes.
CC	[16]	Clear all consecutive counters starting from Counter 0 having their LCE bits cleared; effectively this bit could be used to clear all counters if all of their LCE bits are cleared ■ 0x0 - No effect ■ 0x1 - All counters cleared to zero when this bit is set to 1.
SN	[17]	Take a snapshot of the counters. Bit has write-only property – returns zero on read (RAZ). ■ 0x0 - No effect ■ 0x1 - All counter registers copied into the snapshot registers
Reserved	[15,1], [31:18]	Reserved. Read as zero and ignore on writes.

29.9.4 Interrupt Control Register, CPCT_INT_CTRL

Address: 0xC09

Access: RW

Figure 29-62 CPCT_INT_CTRL



This register enables Interrupt for the respective counter.

Table 29-62 CPCT_INT_CTRL Register Field Description

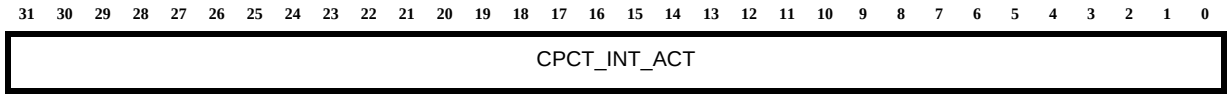
Field	Range	Description
EN	[31:0]	Enable Interrupt for a counter by writing '1' into EN[X] bit where 'X' is the counter number. Writing "1" to a bit position in CPCT_INT_ACT register resetsthe corresponding bit in the CPCT_INT_CTRL register.

29.9.5 Interrupt Active Register, CPCT_INT_ACT

Address: 0xC0A

Access: RW

Figure 29-63 CPCT_INT_ACT



This register indicates all the counters that triggered the PCT Interrupt.

Table 29-63 CPCT_INT_ACT Register Field Description

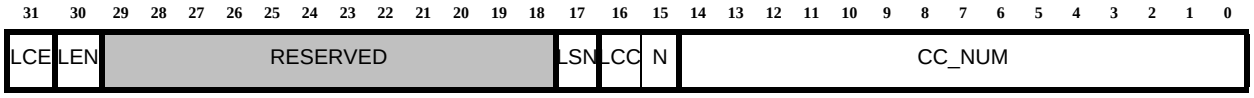
Field	Range	Description
ACT	[31:0]	Interrupt active status. `CPCT_INT_ACT[x]` is set if the counter [x] has met its interrupt condition; assertion of any bit in this register having its corresponding bit in `CPCT_INT_CTRL` will assert an interrupt, which should be connected to an interrupt controller for routing. Writing `1` clears the Interrupt condition to the corresponding PCT counter as well as enable bit for the counter in the `CPCT_INT_CTRL`.

29.9.6 Configuration Register: CPCT_<N>CONFIG

Address: 0xD00

Access: RW

Figure 29-64 CPCT_<N>_CONFIG



Use this register to configure the associated counter.

The condition numbers vary between implementations. Use the countable-conditions registers to determine the allocation of countable-condition name and number on the target counter being accessed.

Table 29-64 CPCT_<N>_CONFIG Register field description

Field	Range	Description
CC_NUM	[14:0]	Countable Condition number
N	[15]	This bit is set to count negated condition, while cleared to count normal condition
LCC	[16]	Clear this counter and subsequent counters having their LCE bits cleared. ■ `0x0`: No effect ■ `0x1`: The count registers are cleared to zero
LSN	[17]	Take a snapshot of this counter and subsequent counters having their LCE bits cleared. ■ `0x0`: No effect ■ `0x1`: the values of count-value registers are copied into the snapshot registers
RESERVED	[29:18]	Reserved. Read as zero and ignore on writes.
LEN	[30]	Enable or disable all counters ■ `0x0`: Disable Counting (Local Disable) ■ `0x1`: Enable Counting (Local Enable)

Table 29-64 CPCT_<N>_CONFIG Register field description

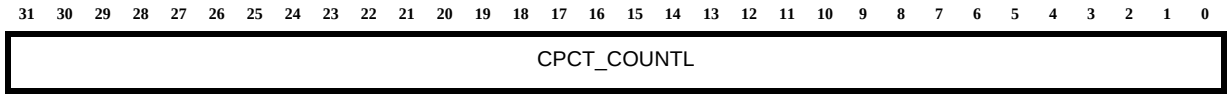
Field	Range	Description
LCE	[31]	Local control enable or disable ■ `0x0`: Local control is inactive, in which the control bits (EN, SN, CC) in this register are ignored and real controls are inherited from the preceding counter; if the preceding counter also has its LC bit cleared, the control is derived from the preceding counter and so on; if counter 0 has its LCE bit cleared, the control is from the global control register (CPCT_CONTROL). ■ `0x1`: Local control is active; the control bits in this register take effect for this counter and other consecutive subsequent counters having their LCE bits cleared.

29.9.7 Count-Value Registers: CPCT_COUNTL

Address: 0xD02

Access: RW

Figure 29-65 CPCT_COUNTL



These registers provides access to the counter selected using CPCT_NUM register. Each count value is an unsigned integer, presented in two 32-bit registers.The lower-order bits are provided in register `CPCT_COUNTL`.

The higher-order bits are provided in register `CPCT_COUNTH`. The higher-order bits are provided in the `CPCT_COUNTH` register. If the counter size is 48 bits, the MSB bits of the `CPCT_COUNTH` register [31:16] are read as zero and ignored on writes. If the counter size is 32 bits, the entire `CPCT_COUNTH` register is read as zero and ignored on writes.

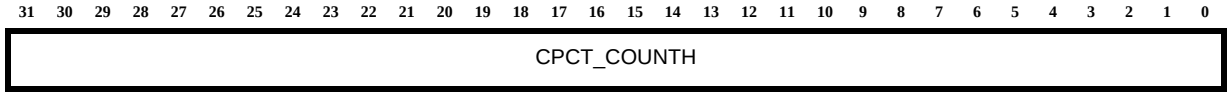
Count values wrap to zero when the maximum count value is reached. If the register is written with a specific value, the counter starts from this value.

29.9.8 Count-Value Registers: CPCT_COUNTH

Address: 0xD03

Access: RW

Figure 29-66 CPCT_COUNTH



These registers provides access to the counter selected using CPCT_NUM register. Each count value is an unsigned integer, presented in two 32-bit registers.The lower-order bits are provided in register `CPCT_COUNTL`.

The higher-order bits are provided in register `CPCT_COUNTH`. The higher-order bits are provided in the `CPCT_COUNTH` register. If the counter size is 48 bits, the MSB bits of the `CPCT_COUNTH` register [31:16] are read as zero and ignored on writes. If the counter size is 32 bits, the entire `CPCT_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.

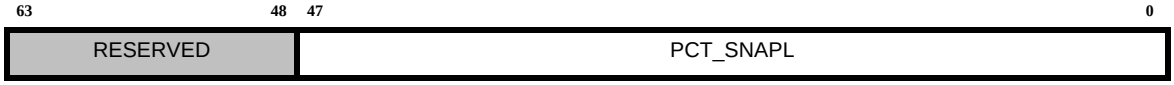
29.9.9 Snapshot-Value Registers, CPCT_<N>_SNAPL

Address: 0xD04

Access: RW

The snapshot feature allows simultaneous capture of all count values without halting ongoing

Figure 29-67 CPCT_<N>_SNAPL Register



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 lower-order bits are provided in the `CPCT_SNAPL` register.
- The higher-order bits are provided in the register `CPCT_SNAPH` register.

Count values from each counter are copied into the associated snapshot registers when a snapshot is triggered through the `CPCT_CONTROL` register. Index selection has no effect on snapshots.

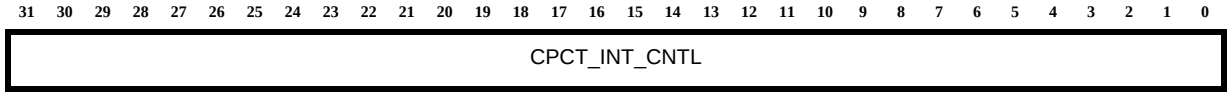
The count value is a 48-bit unsigned integer, presented in 64-bit register PCT_SNAPL. The `PCT_INDEX` register controls access to the counter snapshot. Count values from each counter are copied into the associated snapshot registers when a snapshot is triggered through the `PCT_CONTROL` register. Index selection has no effect on snapshots.

29.9.10 Interrupt Trigger Value Registers, CPCT_INT_CNTL

Address: 0xD06

Access: RW

Figure 29-68 CPCT_INT_CNTL



These registers provide a count value. When a counter reach the count value, an interrupt is triggered if the corresponding counter is configured to raise an interrupt. The counter number to which this interrupt count value applies is controlled by `CPCT_INDEX` register. Each count value is an unsigned integer, presented in two 32-bit registers.

The lower-order bits are provided in register `PCT_INT_CNTL`.

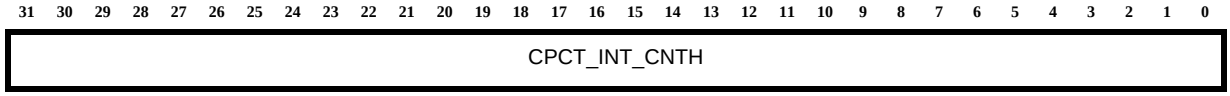
The higher-order bits are provided in register `PCT_INT_CNTH`.

29.9.11 Interrupt Trigger Value Registers, CPCT_INT_CNTH

Address: 0xD07

Access: RW

Figure 29-69 CPCT_INT_CNTH



These registers provide a count value. When a counter reach the count value, an interrupt is triggered if the corresponding counter is configured to raise an interrupt. The counter number to which this interrupt count value applies is controlled by CPCT_INDEX register. Each count value is an unsigned integer, presented in two 32-bit registers.

The lower-order bits are provided in register PCT_INT_CNTL.

The higher-order bits are provided in register PCT_INT_CNTH.

29.9.12 Programming Sequence

The following are the steps in programming to get hard-wired countable condition names for each countable condition number. On a new system, you will need to do this only once.

1. Program CC_NUM in CPCT_CC_NUM register to the countable condition number that you want to count.
2. Read CPCT_CC_NAME0-3 to get the little-endian format of the ASCII string name trailed with zero or in full size.
3. Repeat the preceding steps until names of all the interested countable conditions are retrieved.

The following steps detail the programming and capturing of performance counters using the global control registers when none of the counters is set for local control, or all CPCT_<n>_CONFIG.LCE bits are cleared.

1. Ensure that the counters are stopped for counting by the clearing CPCT_CONTROL.EN bit, and clear all count values by setting CPCT_CONTROL.CC bit.
2. Choose a countable condition for the selected counter by programming the CPCT_<n>_CONFIG.CC_NUM field with an appropriate choice of polarity on CPCT_<n>_CONFIG.N.
3. (Optional) Set an initial count value into the CPCT_<n>_COUNTH and CPCT_<n>_COUNTL registers.
4. (Optional) Set an interrupt triggering value for the counter on CPCT_<n>_INT_CNTH and CPCT_<n>_INT_CNTL registers.
5. Repeat steps 2 through 4 for the counters you intend to use.
6. Set the interrupt enable bits in the CPCT_INT_CTRL register for the counters that are to be configured with interrupt; Ensure that you have cleared all the interrupt enable bits for the counters.
7. Enable counting by setting the CPCT_CONTROL.EN bit.

8. (Optional) Set the CPCT_CONTROL.SN bit to get existing count values of all counters in the corresponding snapshot registers (CPCT_<n>_SNAPH/CPCT_<n>_SNAPL) at any time when some of the values are required.
9. Process interrupts in handler when triggered. In the interrupt handler do the following:
   a. Read the CPCT_INT_ACT register to determine the counters that have raised interrupts.
   b. Write the CPCT_CONTROL.SN bit or the per-counter CPCT_<n>_CONFIG.LSN to snapshot counter values. In such writes also enable the .CC or .LCC bits, so that the counter value issimultaneously reset, thereby clearing the interrupt condition. Taking the snapshot and clearing the counter value is an atomic (simultaneous) action so that counts are not lost.
   c. Read the obtained count values from the snapshot registers. (CPCT_<n>_SNAPH/CPCT_<n>_SNAPL)
   d. Reprogram the CPCT_<n>_INT_CNTH/CPCT_<n>_INT_CNTL registers of selected counters as needed to set new interrupt triggering values.
   e. Write to the CPCT_INT_ACT register to clear interrupt active status, and re-write the enables in the CPCT_INT_CTRL register.
10. Clear the CPCT_CONTROL.EN bit to stop counting.

29.9.13 Programming in a Multi-core Processor

The following are the steps in programming for an administration core to allocate variable number of counters to multiple cores before each of the cores can gain control to use the allocated counters concurrently. For example, allocate x counters to Core C, y counters to Core B, and the rest of counters to Core A, out of a total of *n* counters.

1. Program the CPCT_<n-1-x-1>_CONFIG[LCE]== 1 and all other bits to zero. This programming means that the Counter <n-1-x-1> to Counter <n-1> are controlled together by CPCT_<n-1-x-1>_CONFIG register.
2. Program CPCT_<n-1-x-1-y>_CONFIG[LCE]==1 and all other bits zero. This programming means that the Counter <n-1-x-1-y> to Counter <n-1-x-1-1> are controlled by the CPCT_<n-1-x-1-y>_CONFIG register.
3. Program CPCT_<0>_CONFIG[LCE]==1 and all other bits zero. This programming means that Counter <0> to Counter <n-1-x-1-y-1> are controlled together by the CPCT_<0>_CONFIG register.
4. Pass the base address of Counter 0 registers to Core A, base address of Counter <n-1-x-1-y> registers to Core B, and base address of Counter <n-1-x-1> to Core C, together with the respective numbers of allocated counters.
5. Each core can the set allocated counters corresponding bits in CPCT_INT_CTRL for enabling the interrupt independently.
6. Each core can program registers of the allocated counters appropriately and then start counting by setting the CPCT_<m>_CONFIG.LEN field of the register pointed by the assigned base address, where *m* is the first counter number allocated to the core.
7. Each core can program the CPCT_<m>_CONFIG.LCC and CPCT_<m>_CONFIG.LSN fields for clearing allocated counters or taking a snapshot independently.

8. After a core is done using the counters, the core passes the counter register base address back to the administration core for re-allocation.

Note

The interrupt signal is shared across all cores. Hence, one core must be set to handle the cluster interrupt event and then identify and notify each of Core A, B, or C in case one of assigned counter had contributed to interruption. Upon notification ,each core can access the corresponding bits in the CPCT_INT_ACT/CPCT_INT_CTRL registers and check or re-program the respective counter registers appropriately for handling the interrupt from assinged counters.

30 Hardware Prefetching

Hardware Prefetching is for hiding memory latency. Generally, prefetching is predicting a subsequent memory accesses and pre-fetching it ahead of the actual memory access.

A prefetching mechanism works effectively when:

- Memory addresses are predicted accurately
- The prefetch requests are issued timely, that is: the prefetch data is available when it is needed
- The prefetch data does not replace other useful data in the cache

30.1 Programming Model / Auxiliary Registers

Software interacts with the Hardware Prefetcher through a set of auxiliary registers. The auxiliary registers are as follows:

Address	Auxiliary Register Name	Description
0x4F	HW_PF_CTRL	Hardware Prefetcher Control Auxiliary register

30.1.1 Hardware Prefetcher Control Auxiliary Register, HW_PF_CTRL

Address: 0x4F

Access: RW

Width: 32-bit

Reset: 0x0000_00AA

Figure 30-1 HW_PF_CTRL

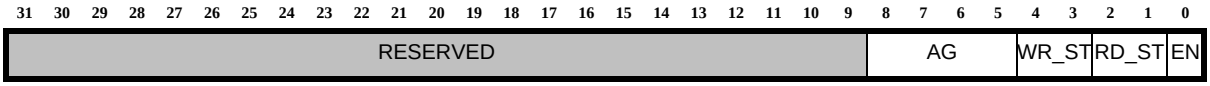


Table 30-1 HW_PF_CTRL Field Description

Field	Bit	Description
AG	[8:5]	Controls the aggressiveness of the hardware prefetching, that is, how much ahead does the core prefetch. The reset value for this field is 5. ■ `0x0`: Non-aggressive prefetch ■ `0x1` to `0x8`: Determines aggressiveness of prefetch ■ `0x9` to `0xF`: Reserved. Any writes greater than 0x8 are clipped to 0x8
WR_ST	[4:3]	This is the write stream threshold. It corresponds to the minimum number of L1 d-cache write misses before the L1 d-cache write policy changes from write-allocate to write-around for the detected write stream. The write stream threshold is defined as 2^{wr_st} for WR_ST != 0. Reset value is `0x1`, that is: a store streaming is triggered after two L1 d-cache misses on a write stream. Programming zero in this field disables the hardware prefetch write streaming regardless of the EN bit.
RD_ST	[2:1]	This is the read stream threshold. It corresponds to the minimum number of L1 d-cache read misses before a hardware prefetching is triggered. The read stream threshold is defined as 2^{rd_st} , for RD_ST != 0. Reset value is `0x1`, that is: a hardware prefetch is triggered after two L1 d-cache misses on a read stream. Programming zero in this field disables the hardware prefetch read streaming regardless of the EN bit.
EN	[0]	Enables Hardware prefetching when asserted. Hardware prefetching is disabled after reset

Note A write to this AUX register is non-serializing.

30.2 Software Prefetching

Software based prefetching does not use any hardware prefetch resource.

30.3 Speculative Accesses

A hardware initiated prefetching is a speculative memory transaction.

In configurations with MMU enabled, a hardware prefetch is only initiated within the same page (for example, 4KB) of the demand load that initiated the hardware prefetch. The ARCv3 cores do not prefetch across MMU page boundaries.

In configurations with MPU enabled, a hardware prefetch is only initiated within the same 16KB range of the demand load that initiated the hardware prefetch.

30.4 Fences and Memory Barriers

A **SYNC** instruction waits until all outstanding hardware prefetches are finished before the SYNC instruction commits.

31 Power Management Features

31.1 Introduction to Power Management Features

The ARCv3 ISA provides a number of features to control power consumption, depending on the configuration of the core. These features may include power domain management (PDM), to reduce static (leakage) power through the power-down of unused components, and features to limit the maximum instruction execution rate of the core to reduce dynamic power (execution control).

- PDM and DVFS Register Interface
- Execution Rate Interface

31.2 PDM and DVFS Register Interface

Table 31-1 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.
0xF7	Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD	Build configuration register

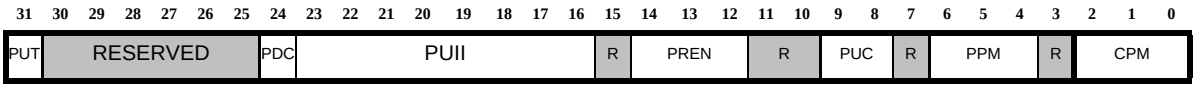
31.2.1 Core Power Status Register, PDM_PSTAT

Address: 0x610

Access: RW

Reset: 0x0

Figure 31-1 PDM_PSTAT Register



This register indicates the current power mode of the core, the previous power mode of the core, the power up cause, and the interrupt index that has caused the power up of the core. The debugger can set the PUT bit of this register to power up the core. If `power_domains` option is `false`, this register is reserved.

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 31-2 PDM_PSTAT Field Description

Field	Bit	Description
CPM	[2:0]	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 up mode ■ `0x1`: PM1 mode ■ `0x2 ~ 0x7`: Reserved For more information about the power-down modes, see Table 31-4 on page 1249.
PPM	[6:4]	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 ~ 0x7`: Reserved For more information about the power-down modes, see Table 31-4 on page 1249.

Table 31-2 PDM_PSTAT Field Description

Field	Bit	Description
PUC	[9:8]	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 *ARCv3-based processor Databook*. ■ `0x1`: Power up by debugger ■ `0x2`: Power up by an external event This field can only be controlled by hardware. Writes to this field are ignored.
PREN	[14:12]	Previous memory power status in power-down mode. Read only. Bit [12]: ICCM previous power status in PM2 mode ■ 0x0: PD2-ICCM shut down ■ 0x1: PD2-ICCM retention Bit [13]: DCCM previous power status in PM2 mode ■ 0x0: PD2-DCCM shut down ■ 0x1: PD2-DCCM retention Bit [14]: Other memory previous power status in PM2 mode ■ 0x0: PD2-OTHERS shut down ■ 0x1: PD2-OTHERS retention In PM1 mode, this field is ignored and return value is 0. This field can only be controlled by hardware. Writes to this field are ignored.
PUII	[23:16]	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]	Power down cause. ■ `0x0`: power down by PMODE register. ■ `0x1`: power down by external event
PUT	[31]	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.

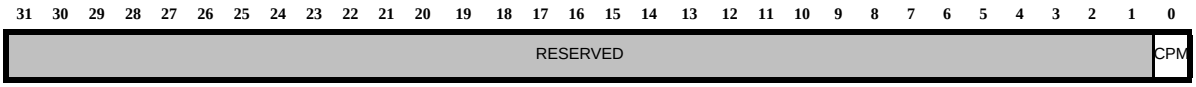
31.2.2 ARC Trace Power Status Register, RTT_PDM_PSTAT

Address: 0x611

Access: RW

Reset: 0x0

Figure 31-2 RTT_PDM_PSTAT Register



Use this register to power down or power up ARC Trace. Reading this register returns the current ARC Trace power status. If `power_domains` option is `false`, this register is reserved.

Table 31-3 RTT_PDM_PSTAT Field Description

Field	Bit	Description
CPM	[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.

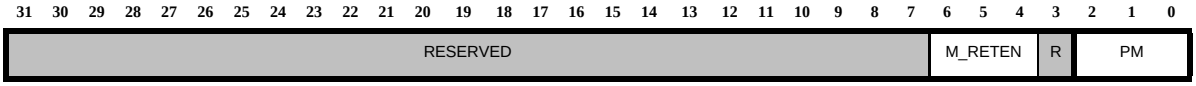
31.2.3 Power Down Register, PDM_PMODE

Address: 0x613

Access: W

Default: 0x0

Figure 31-3 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. If `power_domains` option is `false`, this register is reserved.

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 *ARCv3-based Processor Databook*.

Table 31-4 PDM_PMODE Field Description

Field	Bit	Description
PM	[2:0]	Power-down mode ■ `0x1`: power-down mode 1 (PM1) Other values are reserved
M_RETEN	[6:4]	Memory power down status Bit [4]: ICCM power down status ■ 0x0: PD2-ICCM shut down ■ 0x1: PD2-ICCM retention Bit [5]: DCCM power down status ■ 0x0: PD2-DCCM shut down ■ 0x1: PD2-DCCM retention Bit [6]: Other memory power down status ■ 0x0: PD2-OTHERS shut down ■ 0x1: PD2-OTHERS retention

31.3 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 31-5 Execution Control Registers

Address	Name	Description
0x08	Execution Rate Control Register, EXEC_CTRL	Execution rate control register

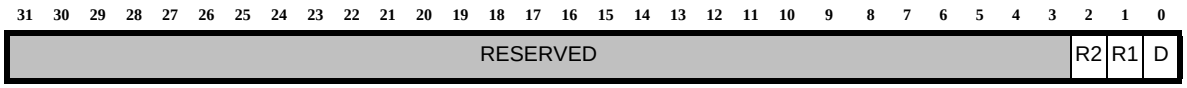
31.3.1 Execution Rate Control Register, EXEC_CTRL

Address: 0x08

Access: RW

Reset 0x0000_0004

Figure 31-4 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`). The R1 and R2 bits control whether their respective feature of RBD is disabled in the micro-architecture. This has no impact on the functional semantics of instruction execution but will have an impact on the cycle-level timing of instruction execution.
The nSIM impact of Relaxed Branch Dispatch is restricted to NCAM modeling of dual dispatch.

Table 31-6 EXEC_CTRL Field Description

Field	Bit	Description
D	[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.
R1	[1]	Specifies if phase 1 of Relaxed Branch Dispatch (RBD) is disabled or enabled: ■ `0x0`: RBD phase 1 is enabled ■ `0x1`: RBD phase 1 is disabled
R2	[2]	Specifies if phase 2 of Relaxed Branch Dispatch (RBD) is disabled or enabled: ■ `0x0`: RBD phase 2 is enabled ■ `0x1`: RBD phase 2 is disabled

31.4 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 ARCv3-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 ARCv3 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.

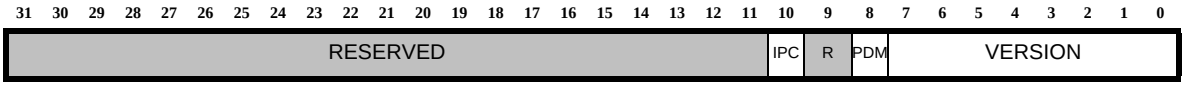
Generally each register has two fields: few least significant bits contain the version number of the extension, and the remaining bits contain configuration information. Any bits within the register that are not required return zero. The version number field is set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCv3-based system.

31.4.1 Power Domain Management and DVFS Build Configuration Register, PDM_DVFS_BUILD

Address: 0xF7

Access: R

Figure 31-5 PDM_DVFS_BUILD Register



This register indicates the presence of the power-domain management capabilities in an ARCv3-based processor.

Note DVFS is not supported.

Table 31-7 PDM_DVFS_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	0x6: current version
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.
IPC	[10]	Indicates whether the IPC (internal power controller) functionality is supported. ■ `0x1`: IPC is supported. ■ `0x0`: IPC is not supported.

32 ARConnect

32.1 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.

32.2 ARConnect Register Interface

Table 32-1 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.
0x603	CONNECT_READBACK_64	Read 64-bit 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_SEMA_BUILD	Build configuration for: ARConnect inter-core semaphore unit
0xD2	CONNECT_MESSAGE_BUILD	Build configuration for: ARConnect inter-core message unit
0xD5	CONNECT_IDU_BUILD	Build configuration for: inter-core interrupt distribution unit

Table 32-1 ARConnect Registers (Continued)

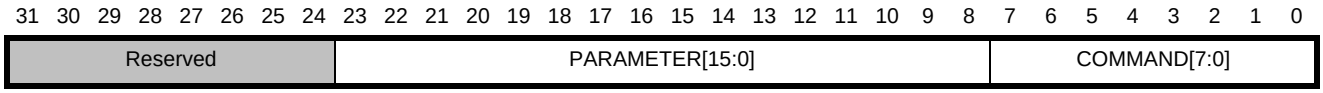
Address	Name	Description
0xD6	CONNECT_GFRC_BUILD	Build configuration for: ARConnect global free running counter
0xE0	CONNECT_ICI_BUILD	Build configuration for: ARConnect inter-core interrupt unit
0xE1	CONNECT_ICD_BUILD	Build configuration for: ARConnect inter-core debug unit

32.2.1 ARConnect Command Register, CONNECT_CMD

Address: 0x600

Access: RW

Figure 32-1 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.

Table 32-2 CONNECT_CMD Register Field Description

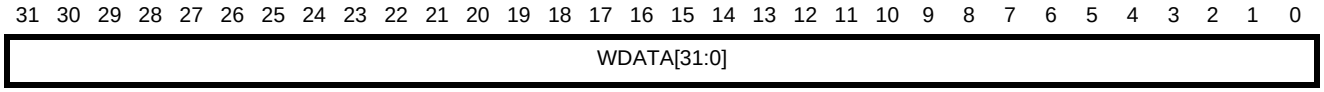
Field	Bits	Description
COMMAND	[7:0]	Command to ARConnect.
PARAMETER	[23:8]	Parameter field of the command to ARConnect. Any parameter that is required by a command is supplied in this field.

32.2.2 ARConnect Write Data Register, CONNECT_WDATA

Address: 0x601

Access: RW

Figure 32-2 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.

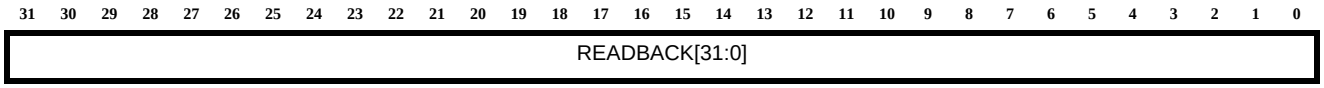
The layout of the CONNECT_WDATA register depends on the ARConnect command being issued.

32.2.3 ARConnect Read Data Register, CONNECT_READBACK

Address: 0x602

Access: R

Figure 32-3 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.

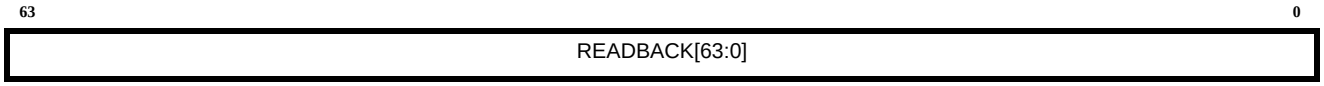
The layout of the CONNECT_READBACK register depends on the issued ARConnect command.

32.2.4 ARConnect Read 64-Bit Data Register, CONNECT_READBACK _64

Address: 0x603

Access: R

Figure 32-4 CONNECT_READBACK_64 Register



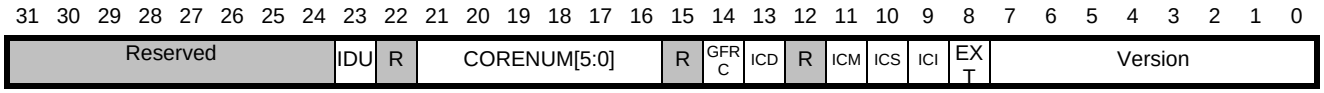
The CONNECT_READBACK_64 register returns 64-bit data when an ARC core issues GFRC read command CMD_GFRC_READ_FULL. This register is read-only; writes to it are ignored and an illegal-instruction exception is raised. You can use the LRL instruction to read this register. If you use the LR instruction, it returns the lower 32 bits to this register.

32.2.5 ARConnect Build Configuration Register, CONNECT_SYSTEM_BUILD

Address: 0xD0

Access: R

Figure 32-5 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.

Field	Bits	Description
VERSION	[7:0]	The ARConnect version. ■ `0x1`: First version ■ `0x2`: Second version; - Changed the starting CORE_ID to 0 from 1 (first version). - Renamed the component from MCIP to ARCONNECT. ■ `0x3`: Current version; indicates that bit [8] is re-purposed for the EXT feature. ASI feature is no longer supported.
EXT	[8]	Indicates whether MCIP can be manipulated through memory-mapped transactions. ■ `0x0`: No peripheral slave port in the HH cluster network. MCIP can not be accessed through memory-mapped transactions. ■ `0x1`:Peripheral slave port is configured in the HH cluster network. MCIP can be accessed through memory-mapped transactions.
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

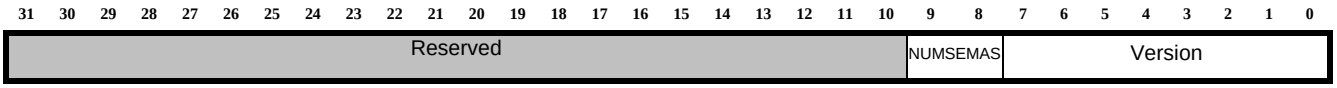
Field	Bits	Description
ICD	[13]	Presence of Inter-Core Debug Unit ■ `0x0`: No inter-core debug unit ■ `0x1`: Has inter-core debug unit
GFRC	[14]	Presence of Global Free-Running Counter Unit ■ `0x0`: No global free-running counter ■ `0x1`: Has global free-running counter
CORENUM	[21:16]	Number of cores (ARC and non-ARC) connected to ARConnect For example 6'h01: 1 core; 6'h02: 2 cores; 6'h04: 4 cores. **Note**: The CORE Number includes all the ARC core and non-ARC cores that have accessibility to MCIP.
IDU	[23]	Presence of Interrupt Distribution Unit ■ `0x0`: No interrupt distribution unit ■ `0x1`: Has interrupt distribution unit

32.2.6 Inter-core Semaphore Unit BCR, CONNECT_SEMA_BUILD

Address: 0xD1

Access: R

Figure 32-6 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.

Table 32-3 CONNECT_SEMA_BUILD Register Field Description

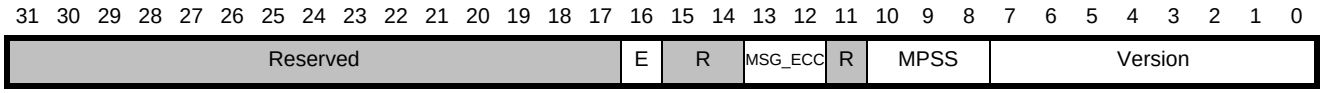
Field	Bits	Description
Version	[7:0]	Inter-core Semaphore Unit version. ■ `0x1`: first version ■ `0x2`: Supports more than 32 accessing clients
NUMSEMAS	[9:8]	Number of Semaphores: ■ `0x0`: 8 semaphores ■ `0x1`: 16 semaphores ■ `0x2`: 32 semaphores ■ `0x3`: reserved

32.2.7 Inter-core Message Unit BCR, CONNECT_MESSAGE_BUILD

Address: 0xD2

Access: R

Figure 32-7 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. This register exists only when `mcip_has_msg_sram==true` in your processor build.

Table 32-4 CONNECT_MESSAGE_BUILD Register Field Description

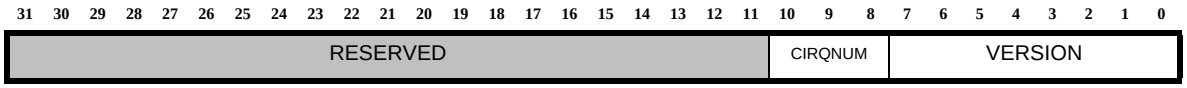
Field	Bits	Description
Version	[7:0]	Inter-core Message Unit version. ■ `0x1`: initial version. ■ `0x2`: Support for ECC protection on SRAM. ■ `0x3`: Added single-bit error counter for ECC.
MPSS	[10:8]	Size of the message-passing SRAM ■ `0x0`: 128 B ■ `0x1`: 256 B ■ `0x2`: 512 B ■ `0x3`: 1 KB ■ `0x4`: 2 KB ■ `0x5`: 4 KB ■ Other values are reserved
MSGECC	[13:12]	ECC protection scheme of Message SRAM ■ `0x0`: No protection ■ `0x1`: ECC (SECDED) protection
ECC_ADDR	[16]	Specifies ECC protection scheme of Message SRAM address ■ `0x0`: No protection ■ `0x1`: ECC protection is configured

32.2.8 Interrupt Distribution Unit BCR, CONNECT_IDU_BUILD

Address: 0xD5

Access: R

Figure 32-8 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 32-5 CONNECT_IDU_BUILD Field Description

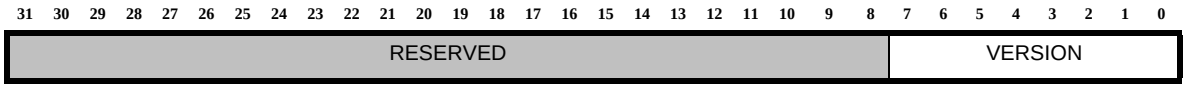
Field	Bit	Description
VERSION	[7:0]	Version number ■ `0x1`: First version ■ `0x2`: Second version; Added software acknowledge mode and support for single core.
CIRQNUM	[10:8]	Number of common interrupts supported in IDU: ■ `0x0` = 4 common interrupts ■ `0x1` = 8 common interrupts ■ `0x2` = 16 common interrupts ■ `0x3` = 32 common interrupts ■ `0x4` = 64 common interrupts ■ `0x5` = 128 common interrupts ■ Other values are reserved

32.2.9 ARConnect Global Free Running Counter BCR, CONNECT_GFRC_BUILD

Address: 0xD6

Access: R

Figure 32-9 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 32-6 CONNECT_GFRC_BUILD Field Description

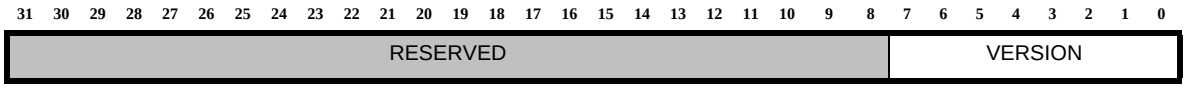
Field	Bit	Description
VERSION	[7:0]	Version number ■ 0x1: First version ■ 0x2: Renamed the global real-time counter (GRTC) to global free running counter (GFRC). ■ 0x3: Added the following commands: CMD_GFRC_SET_CORE, CMD_GRFC_READ_CORE, and CMD_GFRC_READ_HALT. ■ 0x4: Added the following commands: CMD_GFRC_CLK_ENABLE, CMD_GFRC_CLK_DISABLE, CMD_GRFC_READ_CLK_STATUS ■ 0x5: Added the CMD_GFRC_READ_FULL command

32.2.10 Inter-Core Interrupt Unit BCR, CONNECT_ICI_BUILD

Address: 0xE0

Access: R

Figure 32-10 CONNECT_ICI_BUILD Register



The CONNECT_ICI_BUILD register is an optional BCR. If the inter-core interrupt unit is not configured in ARConnect, reading the CONNECT_ICI_BUILD register returns zero.

Table 32-7 CONNECT_ICI_BUILD Field Description

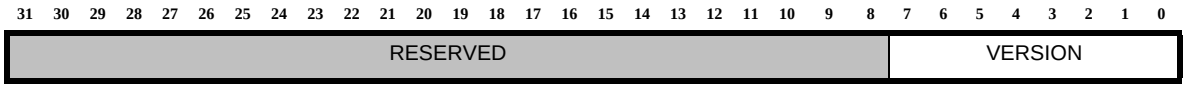
Field	Bit	Description
VERSION	[7:0]	Version number ■ 0x2: This register did not exist in earlier versions of ARConnect (version 0x1). ■ 0x3: Added the following ICI command: CMD_INTRPT_GENERATE_ACK_BIT_MASK. ■ 0x4: Added the following commands: CMD_INTRPT_EXT_MODE, CMD_INTRPT_SET_PULSE_CNT, CMD_INTRPT_READ_PULSE_CNT Updated the following command: CMD_INTRPT_GENERATE_ACK_BIT_MASK

32.2.11 Inter-Core Debug Unit BCR, CONNECT_ICD_BUILD

Address: 0xE1

Access: R

Figure 32-11 CONNECT_ICD_BUILD Register



The CONNECT_ICD_BUILD register is an optional BCR. If the inter-core debug unit is not configured in ARConnect, reading the CONNECT_ICD_BUILD register returns zero.

Table 32-8 CONNECT_ICD_BUILD Field Description

Field	Bit	Description
VERSION	[7:0]	Version number ■ 0x2: First version. This register did not exist in earlier versions of ARConnect.

32.3 Programming Restrictions

32.3.1 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.

32.3.2 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.

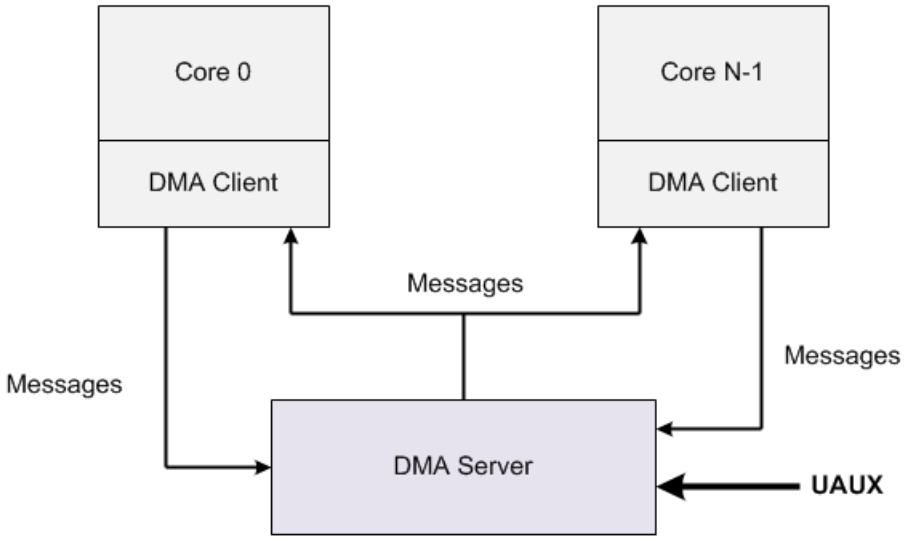
33 Cluster DMA Controller

33.1 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 33-1).

For generic DMA transfers among various memory targets (such as CCMs of ARC core, the Cluster Shared Memory, external peripheral devices, and external memories), the cluster DMA has a single standard AMBA AXI4/ACE-Lite interface to connect with the cluster network DEV port by utilizing the routing capability of the cluster network structure.

Figure 33-1 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 DMA client auxiliary registers
- Trigger message transfer from client to 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 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 data needs to be copied.
- the destination physical byte address, to which 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.

A descriptor is assigned to a channel in the DMA server. Each channel includes a FIFO for storing the 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 back-end 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 33-2 illustrates the system-level diagram of cluster DMA integrated with the cluster network

Figure 33-2 Cluster DMA System Level Architecture

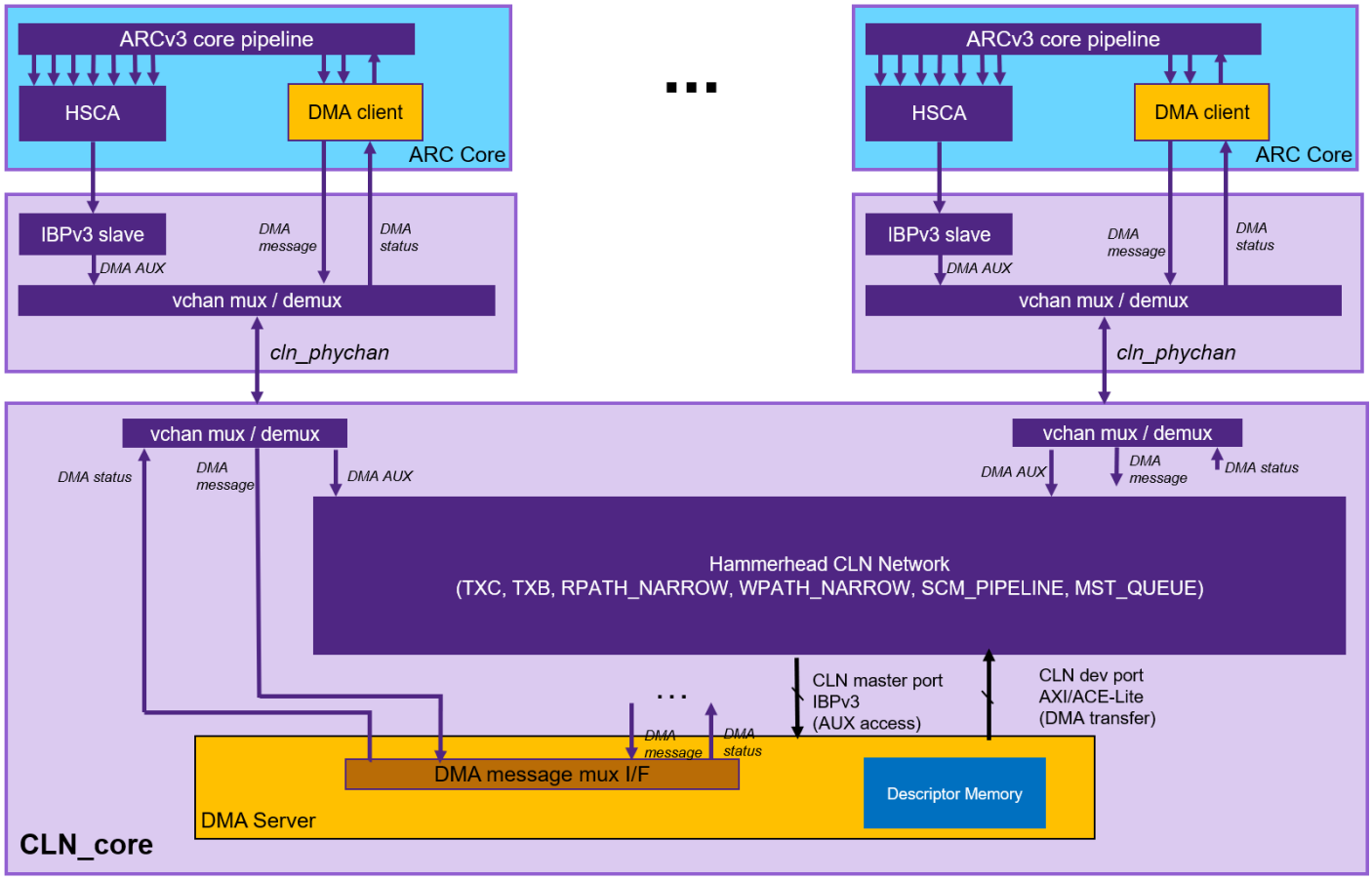
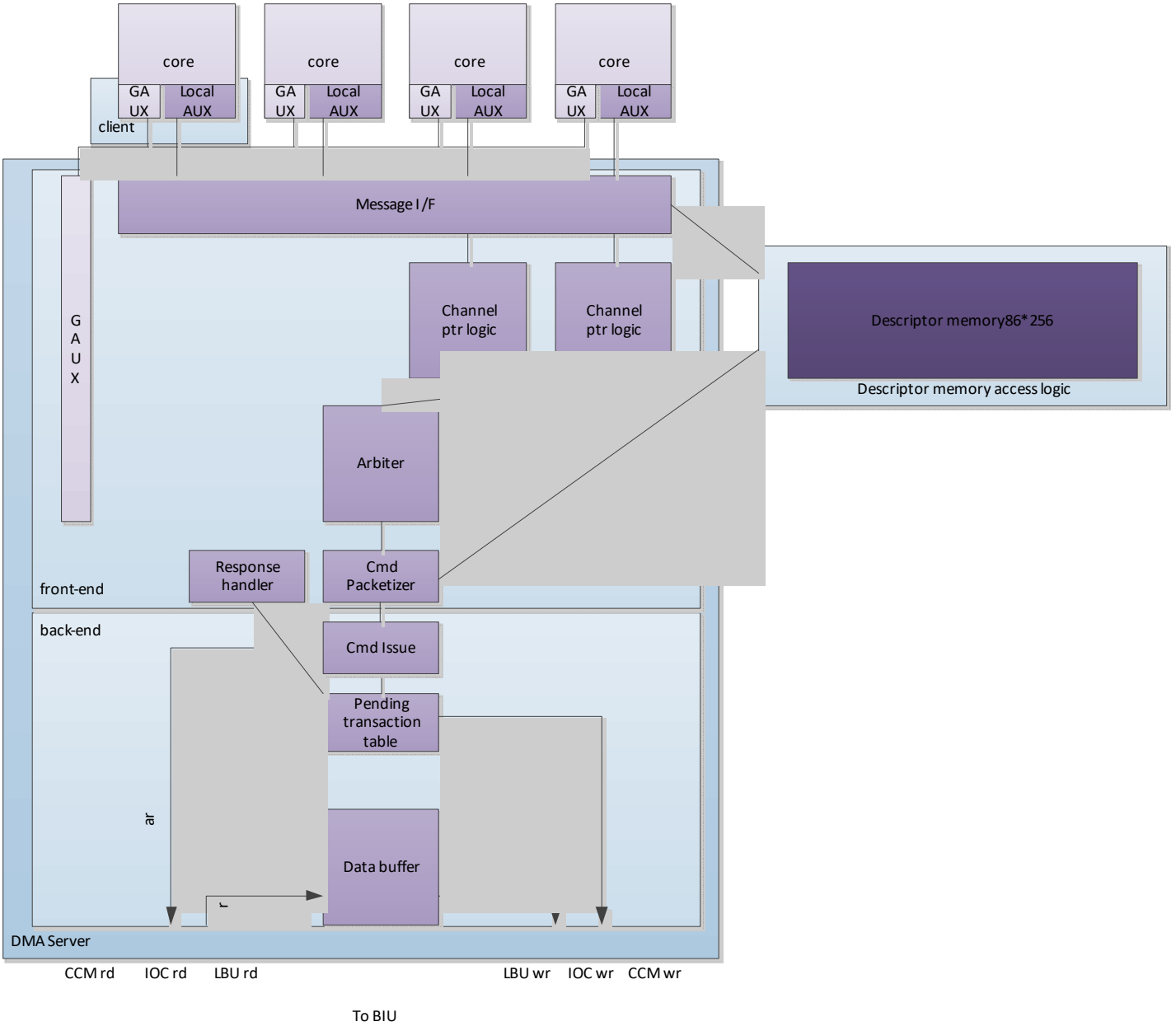


Figure 33-3 on page 1274 illustrates the high-level architecture.

Figure 33-3 Cluster DMA High-Level Architecture



The cluster DMA consists of the following major blocks:

- **DMA client:** logic for issuing and tracking DMA transfers for each core.
- **DMA server front-end:** 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 DMA front-end.
- **DMA server back-end:** logic performing bus transactions.

If a cluster includes a DMA server, each core includes a DMA client that has local auxiliary registers and a UAUX 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 is 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 completion of the DMA transfer.

33.2 Example Use Cases

This section defines some example use cases showing how the cluster DMA controller can be used in an actual application.

The example API defines the following functions, which supports interrupt synchronization, polling-based synchronization, event-based synchronization, and context switching.

- Polling-based synchronization must be used for short DMA transfers.
- Interrupt based synchronization must be used for long DMA transfers due to the higher overhead.

```
// set-up DMA descriptor and hand over to DMA
void dma_start(chan_t    c,    // channel ID, 0 <= c < DMA_SRV_NUM_CHAN
               uint8_t  *from, // from byte address
               uint8_t *to,   // to byte address
               uint17_t  len,  // DMA length in bytes, 1 <= len <= 128K-1
               attr_t    a);   // attributes: (non-)posting, interrupt enable
// set-up DMA descriptor and hand over to DMA, use same channel as previous DMA
void dma_next(uint8_t *from, // from byte address
              uint8_t *to,   // to byte address
              size_t    len,  // DMA length in bytes
              attr_t    a);   // attributes: (non-)posting, interrupt enable
// get handle of most recently started DMA in current context
dma_handle_t dma_get_handle();
// check status of a DMA handle
bool dma_poll_done(dma_handle_t h);
// clear DMA handle done bit
bool dma_clear_done(dma_handle_t h);
// clear the event bit associated with a channel
void dma_clear_event(dma_chan_t c)
// check if DMA client is still busy process dma_start or dma_clear_done
bool dma_check_busy();
// Register a callback that gets executed on an interrupt if handle h is done
void dma_register_cb(void (*foo)(), dma_handle_t h);
// DMA interrupt service routine
void dma_done_isr();
// save/restore context on core context switch
void dma_save_context(uint32_t *s);
```

```
void dma_restore_context(uint32_t *s);
```

33.2.1 Polling Mode

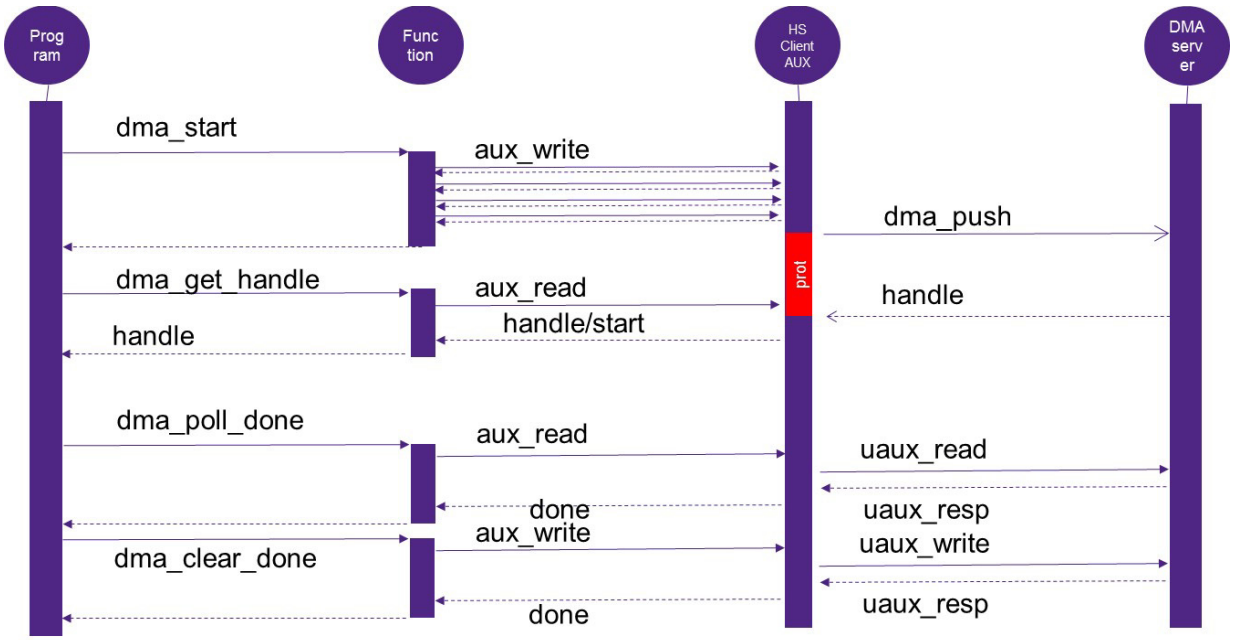
Example 33-1 shows how the DMA controller can be used in a polling mode. The example issues two DMA descriptors into the same channel. The channel processes the DMA descriptors in-order so only the completion of the last DMA descriptors needs to be polled to guarantee the DMA has completed both the transactions.

Example 33-1 Polling-Mode Synchronization

```
// Example doing two DMAs, polling for last one to complete
void dma_example_poll(uint8_t *farr0, // source address for 1st DMA
                      uint8_t *tarr1, // destination address for 1st DMA
                      uint8_t *farr2, // source address for 2nd DMA
                      uint8_t *tarr3, // destination address for 2nd DMA
                      uint17_t len) { // length of DMA transfers
  dma_handle_t h;
  // DMA0: copy farr0 to tarr1 using DMA channel 0,
  // posted write, no interrupt at end
  dma_start(DMA_CHAN0, farr0, tarr1, len, 0);
  // DMA1: start another DMA copy farr2 to tarr3 using DMA channel 0,
  //       non-posted write, poll at end
  dma_next(farr2, tarr3, len, DMA_ATTR_POLL_EN);
  // can do something else here
  // get the handle to ensure last DMA descriptor has been transferred to server
  h = dma_get_handle();
  // poll to check if we are done
  while (!dma_poll_done(h));
  // clear polling flag
  dma_clear_done(h);
  // at this point DMA0 and 1 have completed
}
```

Figure 33-4 shows the sequence of events in polling mode, starting at the second **dma_start()** call. Solid arrows represent requests; dashed arrows represent responses; solid arrow heads represent synchronous communication (blocking); open arrow heads represent asynchronous communication (non-blocking).

Figure 33-4 Polling-Mode Sequence Diagram



33.2.2 Interrupt Mode

Example 33-2 shows how the DMA controller can be used in interrupt mode. Each core has a single interrupt for the DMA client. The example issues two DMA descriptors into the same channel. The channel processes the DMA descriptors in-order, so only the completion of the last DMA raises an interrupt, which triggers a callback function. In this example the POSIX pthreads library is used to unblock the waiting thread.

This example works in an SMP multi-core, preemptive OS context in which each core can access any channel. For bare-metal applications with a fixed channel allocation more simple synchronization can be implemented, for example, based on spinlocks and sleep instructions, and avoiding mutexes and callbacks.

Example 33-2 Interrupt-Mode Synchronization

```
// Define a callback function that gets called by the ISR
void bar() {
  // use pthreads to send event
  pthread_mutex_lock(…);
  pthread_cond_signal(…);
  pthread_mutex_unlock(…);
}

// Actual function to do two DMA, blocking program until the last DMA has completed
void dma_example_interrupt(uint8_t *farr0, // source address for 1st DMA
                           uint8_t *tarr1, // destination address for 1st DMA
                           uint8_t *farr2, // source address for 2nd DMA
                           uint8_t *tarr3, // destination address for 2nd DMA
                           uint17_t len) { // length of DMA transfers
  dma_handle_t h;
  // Two DMA transactions
```

```
  dma_start(DMA_CHAN0, farr0, tarr1, len, 0);
  dma_next(farr2, farr3, len, DMA_ATTR_INT_EN);
  // get the handle to ensure last DMA descriptor
  // has been transferred to server
  h = dma_get_handle();
  // lock a mutex
  pthread_mutex_lock(…);
  // now wait for the DMA to complete; sleep until ISR wakes us up
  dma_register_cb(&bar, h);
  // wait for callback to send some event
  pthread_cond_wait(…);
  pthread_mutex_unlock(…);
  // Both DMAs are done
}
```

33.2.3 Event Mode

In an event-synchronization, the Cluster DMA can use a simplified synchronization scheme. In this simplified synchronization scheme the DMA supports one wake-up event for each DMA channel for each core. 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, the following procedure is used:

1. The core issues a new DMA descriptor while setting the EVENT attribute in the descriptor.
2. The core issues a wait-for-event instruction (WEVT).
3. Upon the completion of the DMA transaction, the server triggers a wake up event in the originating CPU.

Example 33-3 Event-Mode Synchronization

```
// Example doing two DMAs, sleeping until the last DMA completes
void dma_example_baremetal(uint8_t *farr0, // source address for 1st DMA
                           uint8_t *tarr1, // destination address for 1st DMA
                           uint8_t *farr2, // source address for 2nd DMA
                           uint8_t *tarr3, // destination address for 2nd DMA
                           uint17_t len) { // length of DMA transfers
  dma_handle_t h;
  // DMA0: copy farr0 to tarr1 using DMA channel 0,
  // posted write, no interrupt at end
  dma_start(DMA_CHAN0, farr0, tarr1, len, 0);
  // DMA1: start another DMA copy farr2 to tarr3 using DMA channel 0,
  //       non-posted write, poll at end
  dma_next(farr2, tarr3, len, DMA_ATTR_EVENT_EN);
  // can do something else here
  // get the handle to ensure last DMA descriptor has been transferred to server
  h = dma_get_handle();
  // go to sleep, waiting for event to trigger wake-up
  do {
   arc_wevt();   // ARC intrinsic call: sleep while waiting for a wake-up event
   // check if this is a spurious wake-up
  } while (!dma_poll_done(h));
```

```
  // clear polling flag
  dma_clear_done(h);
  // clear wake-up flag for corresponding channel
  dma_clear_event(DMA_CHAN0);
  // at this point DMA0 and 1 have completed
}
```

33.2.4 Low level API

This section shows an example implementation of the low-level API functions as used in the examples.

33.2.4.1 dma_start()

The **dma_start()** function pushes a new descriptor into a channel.

```
// function to start DMA (non-blocking)
// duration: 5 cycles
void dma_start(dma_chan_t c,       // channel ID, 0 <= c < DMA_NUM_CHAN
               dma_addr_t *src,    // from byte address
               dma_addr_t *dst,    // to byte address
               dma_len_t   len,    // DMA length in bytes
               dma_attr_t  a) {    // attributes, including START bit
  // Write will block iff busy i.e. DMA_C_STAT_AUX.B bit is set
  aux_wr(c, DMA_C_CHAN_AUX);
  // Rest is same as dma_next
  dma_next(src, dst, len, a);
}
```

33.2.4.2 dma_next()

This function pushes a new DMA descriptor without selecting a channel explicitly, assuming that the channel-selection register is already set up.

```
// function to start DMA (non-blocking)
// duration: 4 cycles
void dma_next(dma_addr_t *src,    // from address
              dma_addr_t *dst,    // to address
              dma_len_t   len,    // DMA length in bytes
              dma_attr_t  a) {    // attributes, including START bit
  // Write will block iff busy i.e. DMA_C_STAT_AUX.B bit is set
  aux_wr(src, DMA_C_SRC_AUX);
  aux_wr(dst, DMA_C_DST_AUX);
  aux_wr(a, DMA_C_ATTR_AUX);
  // the LENGTH register triggers the actual dma_push message
  aux_wr(len, DMA_C_LEN_AUX);
}
```

33.2.4.3 dma_get_handle()

```
// function to get handle of last started DMA
// duration: 1 cycle
dma_handle_t dma_get_handle(dma_handle_t h) {
 // AUX read will block iff busy i.e. DMA_C_STAT_AUX.B bit is set
 return aux_rd(DMA_C_HANDLE_AUX);
}
```

33.2.4.4 dma_poll_busy()

This function can be used to avoid stalling when pushing series of DMA descriptors.

```
// function to check if DMA is busy sending descriptor
// duration: 1 cycle
bool dma_poll_busy() {
 return aux_rd(DMA_C_STAT_AUX);
}
```

33.2.4.5 dma_clear_done()

```
// function to clear done bit of a handle
// duration: 12 cycle
void dma_clear_done(dma_handle_t h) {
  uint32 a;     // index of status register
  uint32 o;     // bit offset in status register
  bool   s;     // true/false result
  a = h>>5;     // divide by 32
  o = (1 << (h & 0x1f)); // one-hot bitmask
  aux_wr(o, DMA_S_DONE0_AUX+a);
```

33.2.4.6 dma_clear_event()

```
// function to clear a channel event bit
// duration: 2 cycles
void dma_clear_event(dma_chan_t c) {
  aux_wr(1<<c, DMA_S_EVSTAT_CLR_AUX);
}
```

33.2.4.7 dma_register_cb()

The interrupt handler uses a table of callback functions, with one table entry for each descriptor handle. In this example, accesses to the table are protected by a pthreads mutex to guarantee atomic accesses to the table and the done bits. Alternatively, interrupts may be disabled, avoiding mutex overhead.

```
// table of callbacks for ISR, volatile
// a mutex in shared memory protects access to the table and the DMA server done bits
// the table and mutex are shared across processes on a single core
// and should be shared across cores in SMP configurations
void (*dma_cb_table[DMA_DECR_NUM])();
pthread_mutex_t dma_cb_table_mutex = PTHREAD_MUTEX_INITIALIZER;

void dma_register_cb(void (*foo)(), dma_handle_t h) {
 // lock mutex associated with callback table
 pthread_mutex_lock(&dma_cb_table_mutex);
 // check if DMA is already done
 d = dma_poll_done(h);
 if (d) {
  // already done, callback immediately
  dma_clear_done(h);
  pthread_mutex_unlock(&dma_cb_table_mutex);
  (*foo)();
 } else {
  // not done, register callback
```

```
  dma_cb_table[h] = foo;
  pthread_mutex_unlock(&dma_cb_table_mutex);
 }
}
```

33.2.4.8 dma_done_isr()

This function handles the done interrupt. The ISR calls back to functions registered in a table by **dma_register_cb()**. In this example ISR the callback table and done bit access are protected by a pthreads mutex to guarantee atomicity.

```
// handle DMA interrupts (user mode part)
void dma_done_isr() {
  // cycle through all DONE bits in groups of 32
  int h = 0; // descriptor counter
  pthread_mutex_lock(&dma_cb_table_mutex);
  for (int i = 0; i < DMA_NUM_DESCR/32; i++) {
    // read done status
    stat = aux_rd(DMA_S_DONESTAT0_AUX + i);
    if (stat != 0) {
      // at least 1 out of 32 DONE bits is set
      int c = 0; // store done bits that need to be cleared
      // loop over all 32 bits in the DONE status
      for (int j = 0; j < 32; j++, h++, stat >>= 1) {
        c = c >> 1;
        if ((stat & 1) && (dma_cb_table[h] != NULL)) {
          // found a set bit with a valid callback
          // execute callback and remove
          *dma_cb_table[h]();
          dma_cb_table[h] = NULL;
          // remember bit so we can clear it later
          c |= 0x80000000;
        }
      }
      if (c != 0) {
        // clear DONE bits that were handled
        aux_wr(c, DMA_S_DONESTAT0_CLR_AUX + i);
      }
    }
  }
  pthread_mutex_unlock(&dma_cb_table_mutex);
}
```

33.2.5 Context Switching

The following functions can be used to implement safe context switches in a preemptive OS environment.

```
// save core specific DMA context
void save_dma_context(uint32_t *s) {
   // DMA AUX regs in context
   // read access will block iff DMA_C_STATUS.B is set
   *s++ = aux_rd(DMA_C_HANDLE_AUX);
   *s++ = aux_rd(DMA_C_CHAN_AUX);
   *s++ = aux_rd(DMA_C_SRC_AUX);
```

```
    *s++ = aux_rd(DMA_C_DST_AUX);
    *s++ = aux_rd(DMA_C_ATTR_AUX);
}
// restore core specific DMA context
void restore_dma_context(uint32_t *s) {
          aux_wr(*s++, DMA_C_HANDLE_AUX);
          aux_wr(*s++, DMA_C_CHAN_AUX);
          aux_wr(*s++, DMA_C_FROM_AUX);
          aux_wr(*s++, DMA_C_TO_AUX);
          aux_wr(*s++, DMA_C_ATTR_AUX);
}
```

33.3 Configuration Options

For notes about the configuration options and their dependencies, see the *ARC HS Series Databook*.

33.4 Programming the Cluster DMA using Auxiliary Registers

The cluster DMA controller auxiliary registers are categorized into the following:

- Build configuration register "DMA 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 33-1 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 33-1 Cluster DMA Auxiliary Register List

Address	Description
0x0E6	DMA Build Configuration Register, DMA_BUILD
Cluster DMA Client AUX Registers	
0xD00	DMA Client Control Register, DMA_C_CTRL_AUX
0xD01	DMA Client Channel Select Register, DMA_C_CHAN_AUX
0xD02	DMA Client Source Address Register, DMA_C_SRC_AUX
0xD04	DMA Client Destination Address Register, DMA_C_DST_AUX
0xD06	DMA Client Attributes Register, DMA_C_ATTR_AUX
0xD07	DMA Client Length Register, DMA_C_LEN_AUX

Table 33-1 Cluster DMA Auxiliary Register List

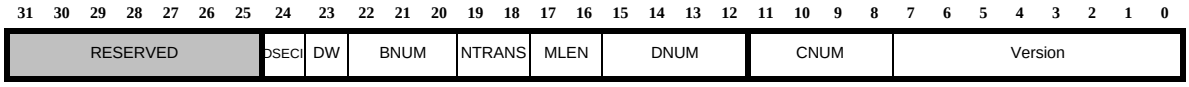
0xD08	DMA Client Handle Register, DMA_C_HANDLE_AUX
0xD0A	DMA Event Status Register, DMA_C_EVSTAT_AUX
0xD0B	DMA Client Clear Event Status Register, DMA_C_EVSTAT_CLR_AUX
0xD0C	DMA Client Status Register, DMA_C_STAT_AUX
0xD0D	DMA Client Interrupt Status Register, DMA_C_INTSTAT_AUX
0xD0E	DMA Client Clear Interrupt Status Register, DMA_C_INTSTAT_CLR_AUX
0xD0F	DMA Client Error Handle Register, DMA_C_ERRHANDLE_AUX
Cluster DMA Server AUX Registers	
0xD10	DMA Server Control Register, DMA_S_CTRL_AUX
0xD20+D	DMA_S_DONESTATD_AUX
0xD40+D	DMA Server Clear Done Status Register, DMA_S_DONESTATD_CLR_AUX
0xD80+(C*8)	DMA Server Channel Tail Pointer Register, DMA_S_TAILC_AUX
0xD81+(C*8)	DMA Server Channel Middle Pointer Register, DMA_S_MIDC_AUX
0xD82+(C*8)	DMA Server Channel Head Pointer Register, DMA_S_HEADC_AUX
0xD83+(C*8)	DMA Server Channel Base Register, DMA_S_BASEC_AUX
0xD84+(C*8)	DMA Server Channel Last Register, DMA_S_LASTC_AUX
0xD85+(C*8)	DMA Channel Priority Register, DMA_S_PRIORC_AUX
0xD86+(C*8)	DMA Channel Control Register, DMA_S_STATC_AUX

33.4.1 DMA Build Configuration Register, DMA_BUILD

Address: 0x0E6

Access: R

Figure 33-5 DMA_BUILD Register



The DMA_BUILD register contains the version and configuration information of the Cluster DMA.

Table 33-2 DMA_BUILD Field Description

Field	Bit	Description
Version	[7:0]	■ 0x10: Supports access all ARC CCM (ICCM/DCCM)/Cluster Shared Memory/external memory, and shared cache pre-loading
CNUM	[11:8]	Specifies the number of channels in the DMA server $\log_2$(DMA_SRV_NUM_CHAN)
DNUM	[15:12]	Specifies the number of descriptors in the DMA server $\log_2$(DMA_SRV_NUM_DESCR)
MLEN	[17:16]	Specifies the maximum burst size for a bus transaction $\log_2$(DMA_SRV_MAX_BURST/4)
NTRANS	[19:18]	Specifies the maximum number of pending bus transactions $\log_2$(DMA_SRV_MAX_TRANS/4)
BNUM	[22:20]	Specifies the DMA server data buffer size in data elements. $\log_2$(DMA_SRV_BUF_SIZE/16) ■ x000: 16 ■ x001: 32 ■ x010: 64 ■ x011: 128 ■ x100: 256 ■ x101: 384
DW	[23]	Specifies the internal data width of the DMA server. $\log_2$(DMA_SRV_DATA_SIZE/64) Fixed to 1'b1 for 128-bit buffer size

Table 33-2 DMA_BUILD Field Description

Field	Bit	Description
DESCI	[24]	Implementation of DMA descriptor memory (`DMA_SRV_DESC_RAM_IMPLEMENTATION`) `0x0`: Flip-flop `0x1`: Memory

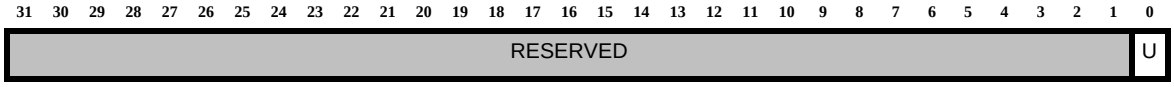
33.4.2 DMA Client Control Register, DMA_C_CTRL_AUX

Address: 0xD00

Access: RW

Reset: 0x0

Figure 33-6 DMA_C_CTRL_AUX Register



Use this register to control access to the DMA client registers in kernel mode and user mode.

Table 33-3 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.

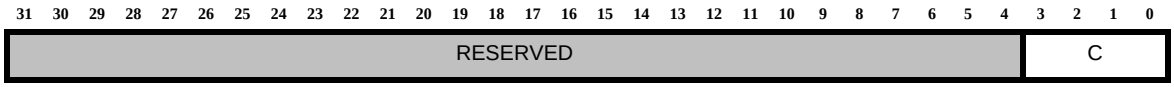
33.4.3 DMA Client Channel Select Register, DMA_C_CHAN_AUX

Address: 0xD01

Access:
- RW when `DMAC_C_CTRL_AUX.U==0`
- rw when `DMAC_C_CTRL_AUX.U==1`

Reset: 0x0

Figure 33-7 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 33-4 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: $C = \log_2(DMA\_NUM\_CHAN)$

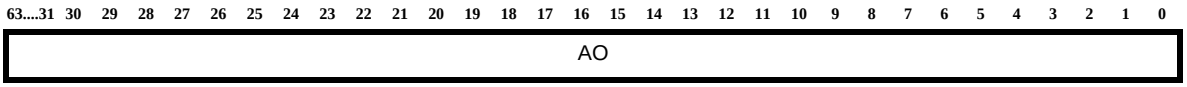
33.4.4 DMA Client Source Address Register, DMA_C_SRC_AUX

Address: 0xD02

Access:
- RW when `DMAC_C_CTRL_AUX.U==0`
- rw when `DMAC_C_CTRL_AUX.U==1`

Reset: 0x0

Figure 33-8 DMA_C_SRC_AUX Register



Use this register to specify the source byte address for the DMA descriptor. The register width of DMA_C_SRC_AUX is 64-bit.

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 33-5 DMA_C_SRC_AUX Field Description

Field	Bit	Description
ALO	[31:0]	DMA client source byte address for the DMA descriptor

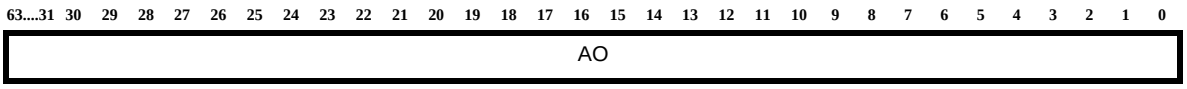
33.4.5 DMA Client Destination Address Register, DMA_C_DST_AUX

Address: 0xD04

Access:
- RW when `DMAC_C_CTRL_AUX.U==0`
- rw when `DMAC_C_CTRL_AUX.U==1`

Reset: 0x0

Figure 33-9 DMA_C_DST_AUX Register



Use this register to specify the destination byte address for the DMA descriptor. The register width of DMA_C_DST_AUX is 64-bit.

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 33-6 DMA_C_DST_AUX Field Description

Field	Bit	Description
ALO	[31:0]	DMA client destination byte address for the DMA descriptor

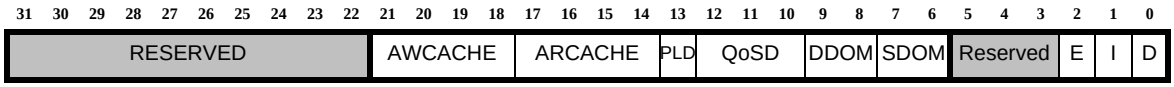
33.4.6 DMA Client Attributes Register, DMA_C_ATTR_AUX

Address: 0xD06

Access:
- RW when `DMAC_C_CTRL_AUX.U==0`
- rw when `DMAC_C_CTRL_AUX.U==1`

Reset: 0x88000

Figure 33-10 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.

Note

- Cluster DMA only supports transaction with modifiable attribute, that is, all DMA transactions initiated by DMA server have ARCACHE[1]/AWCACHE[1] == 1'b1. Write to DMA_C_ATTR_AUX[15] or DMA_C_ATTR_AUX[19] with zero is ignored; Read value of DMA_C_ATTR_AUX[15] or DMA_C_ATTR_AUX[19] is always one.
- If the source or destination targets peripheral regions in the system, ARCACHE[3:2]/AWCACHE[3:2] should be programmed to 2'b00, SDMO/DDOM should be programmed to 2'b11 (system attribute).

Table 33-7 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`).

Table 33-7 DMA_C_ATTR_AUX Field Description

Field	Bit	Description
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).
SDOM	[7:6]	Specify the domain attributes of DMA transactions to access source memory as follows: ■ 2'b00 (non-shareable): Data in this region can be cached but is considered private, and hence no snooping occurs to addresses in this region. ■ 2'b01 (Inner-shareable): Data in this region is considered to be shareable within the local cluster. The region is included in the local (or inner) coherency domain, and snooping is limited to this inner domain. ■ 2'b10 (Outer-shareable): Data in this region is considered to be shareable across the system. The region is included in the global coherency domain, and snoop requests are propagated through the inter-cluster interconnect to maintain global coherency. ■ 2'b11 (System): Data in this region can be cached but is considered private, and hence no snooping occurs to addresses in this region. Issue coherency snoop transactions when this field is set to 2'b01 or 2'b10.
DDOM	[9:8]	Specify the domain attributes of DMA transactions to access destination memory as follows: ■ 2'b00 (non-shareable): Data in this region can be cached but is considered private, and hence no snooping occurs to addresses in this region. ■ 2'b01 (Inner-shareable): Data in this region is considered to be shareable within the local cluster. The region is included in the local (or inner) coherency domain, and snooping is limited to this inner domain. ■ 2'b10 (Outer-shareable): Data in this region is considered to be shareable across the system. The region is included in the global coherency domain, and snoop requests are propagated through the inter-cluster interconnect to maintain global coherency. ■ 2'b11 (System): Data in this region can be cached but is considered private, and hence no snooping occurs to addresses in this region. Issue coherency snoop transactions when this field is set to 2'b01 or 2'b10.
QoSD	[12:10]	Specify QoS resource domain for each DMA descriptor used for read/write bus transactions from source to destination memories. The QoS resource domain information is handled inside the cluster network (TXC) to provide better Quality of Service to shared resources.
PLD	13	Set this bit to enable Shared Cache Preloading. For more information, see *Shared Cache Preloading* in the Databook.

Table 33-7 DMA_C_ATTR_AUX Field Description

Field	Bit	Description
ARCACHE	[17:14]	Specify the attributes for DMA read transactions
		[14]: bufferable ■ 0: non bufferable ■ 1: bufferable
		[15]: modifiable (cacheable) ■ 0: non cacheable ■ 1: cacheable
		[16]: allocate (read allocate) ■ 0: no read allocate ■ 1: read allocate
		[17]: other allocate (write allocate) ■ 0: no write allocate ■ 1: write allocate
AWCACHE	[21:18]	Specify the attributes for DMA write transactions
		[18]: bufferable 0: non bufferable 1: bufferable
		[19]: modifiable (cacheable) 0: non cacheable 1: cacheable
		[20]: other allocate (read allocate) 0: no read allocate 1: read allocate
		[21]: allocate (write allocate) 0: no write allocate 1: write allocate

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.

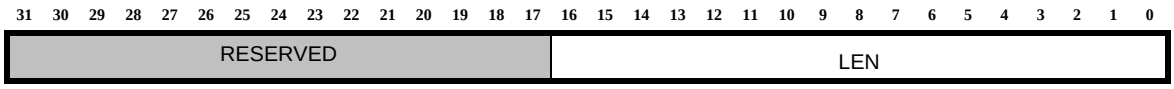
33.4.7 DMA Client Length Register, DMA_C_LEN_AUX

Address: 0xD07

Access:
- rW when DMAC_C_CTRL_AUX.U==0
- rw when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 33-11 DMA_C_LEN_AUX Register



Use this register to specify the length of the DMA transfer in bytes. The maximum DMA transfer size is 2^17 - 1 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 33-8 DMA_C_LEN_AUX Field Description

Field	Bit	Description
LEN	[16:0]	Specifies the length of a DMA transfer in bytes.

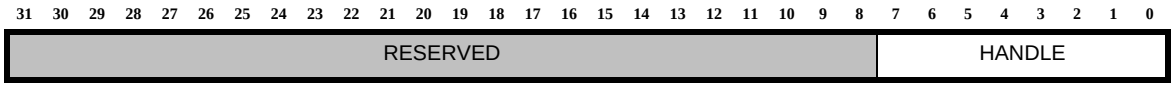
33.4.8 DMA Client Handle Register, DMA_C_HANDLE_AUX

Address: 0xD08

Access:
- RW when DMAC_C_CTRL_AUX.U==0
- rW when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 33-12 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 33-9 DMA_C_HANDLE_AUX Field Description

Field	Bit	Description
HANDLE	[7:0]	Specifies the descriptor index of the most recently pushed DMA descriptor.

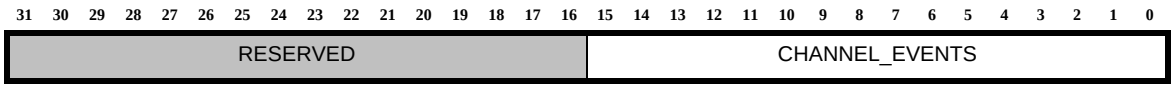
33.4.9 DMA Event Status Register, DMA_C_EVSTAT_AUX

Address: 0xD0A

Access: r

Reset: 0x0

Figure 33-13 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 33-10 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.

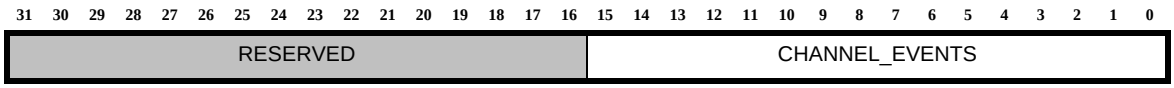
33.4.10 DMA Client Clear Event Status Register, DMA_C_EVSTAT_CLR_AUX

Address: 0xD0B

Access: rW

Reset: 0x0

Figure 33-14 DMA_C_EVSTAT_CLR_AUX Register



Use this register to clear the DMA channel event status for the core.

Table 33-11 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.

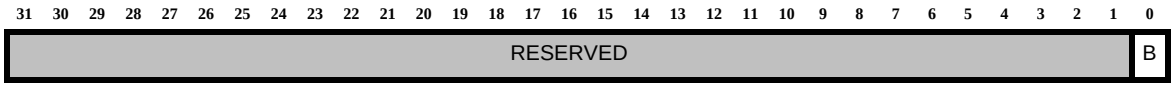
33.4.11 DMA Client Status Register, DMA_C_STAT_AUX

Address: 0xD0C

Access:
- R
- r when DMAC_C_CTRL_AUX.U==1

Reset: 0x0

Figure 33-15 DMA_C_STAT_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 33-12 DMA_C_STAT_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.

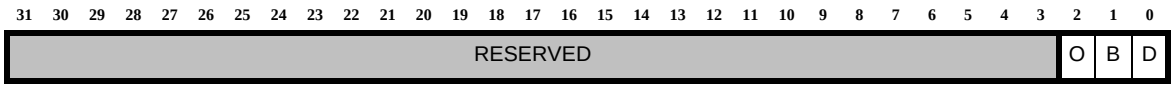
33.4.12 DMA Client Interrupt Status Register, DMA_C_INTSTAT_AUX

Address: 0xD0D

Access: R

Reset: 0x0

Figure 33-16 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 33-13 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 33-13 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.

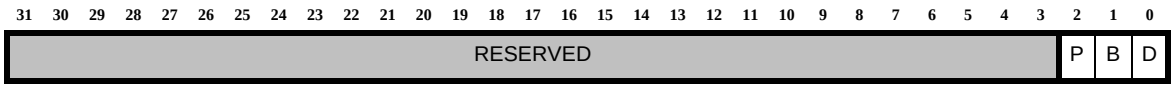
33.4.13 DMA Client Clear Interrupt Status Register, DMA_C_INTSTAT_CLR_AUX

Address: 0xD0E

Access: RW

Reset: 0x0

Figure 33-17 DMA_C_INTSTAT_CLR_AUX Register



Use this register to clear the DMA interrupt status for the core.

Table 33-14 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.

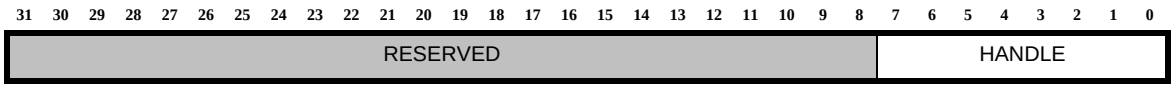
33.4.14 DMA Client Error Handle Register, DMA_C_ERRHANDLE_AUX

Address: 0xD0F

Access: R

Reset: 0x0

Figure 33-18 DMA_C_ERRHANDLE_AUX Register



This register contains the handle, the descriptor index, that caused the most recent bus error.

Table 33-15 DMA_C_ERRHANDLE_AUX Field Description

Field	Bit	Description
HANDLE	[7:0]	Specifies the descriptor index that caused the most recent bus error.

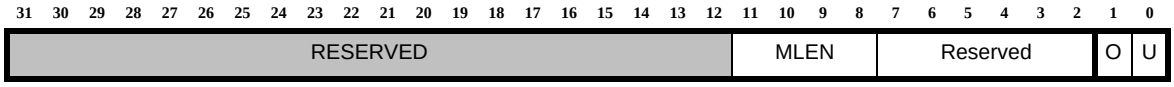
33.4.15 DMA Server Control Register, DMA_S_CTRL_AUX

Address: 0xD10

Access: RW

Reset: 0x0

Figure 33-19 DMA_S_CTRL_AUX Register



This register controls the pseudo-static parameters for the DMA server.

Table 33-16 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.

MLEN	[11:8]	Burst length
		`0<= MLEN <=` $\log_2$`(DMA_MAX_BURST)`
		Specifies the maximum bus burst length as a $\log_2$.
		Effective maximum bus burst length is:
		`min(2`$^{\texttt{DMA\_S\_CTRL\_AUX.MLEN}}$`, DMA_SRV_MAX_BURST).`

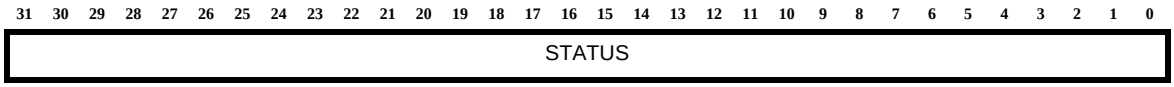
33.4.16 DMA Server Done Status Register, DMA_S_DONESTATD_AUX

Address: 0xD20+D

Access:
- R when DMAC_S_CTRL_AUX.U==0
- r when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 33-20 DMA_S_DONESTATD_AUX Register



The DMA_S_DONESTAT*D*_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_DONESTAT*D*_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 33-17 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

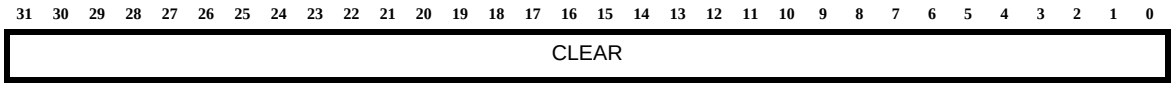
33.4.17 DMA Server Clear Done Status Register, DMA_S_DONESTATD_CLR_AUX

Address: 0xD40+D

Access:
- RW when DMAC_S_CTRL_AUX.U==0
- rw when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 33-21 DMA_S_DONESTATD_AUX Register



The DMA_S_DONESTAT*D*_CLR_AUX register is used to clear the status bits (DMA_S_DONESTAT*D*_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 33-18 DMA_S_DONESTATD_AUX Field Description

Field	Bit	Description
CLEAR	[31:0]	Clear the status bits (DMA_S_DONESTAT*D*_AUX) for the descriptors: *D**32 through *D**32+31.

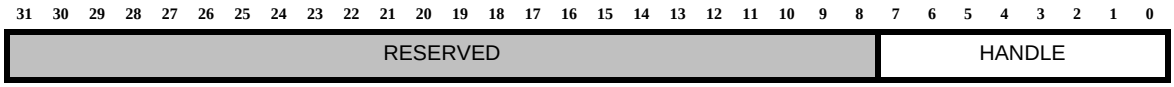
33.4.18 DMA Server Channel Tail Pointer Register, DMA_S_TAILC_AUX

Address: 0xD80+(C*8)

Access:
- R when DMAC_S_CTRL_AUX.U==0
- r when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 33-22 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.

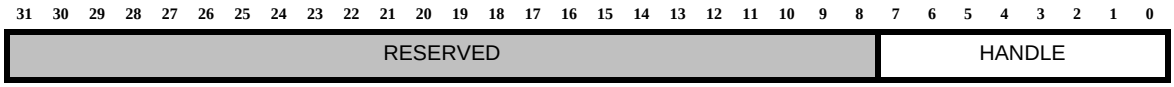
33.4.19 DMA Server Channel Middle Pointer Register, DMA_S_MIDC_AUX

Address: 0xD81+(C*8)

Access:
- R when DMAC_S_CTRL_AUX.U==0
- r when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 33-23 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.

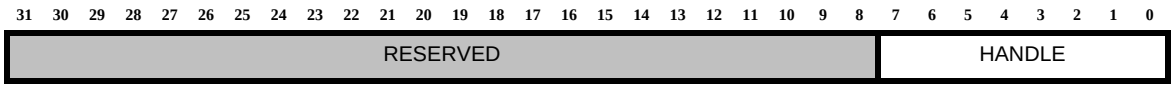
33.4.20 DMA Server Channel Head Pointer Register, DMA_S_HEADC_AUX

Address: 0xD82+(C*8)

Access:
- R when DMAC_S_CTRL_AUX.U==0
- r when DMAC_S_CTRL_AUX.U==1

Reset: 0x0

Figure 33-24 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.

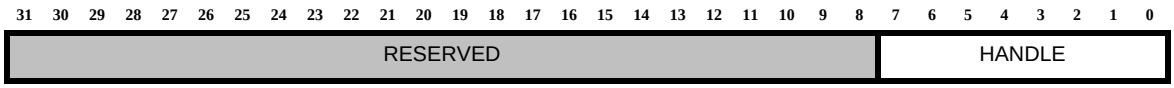
33.4.21 DMA Server Channel Base Register, DMA_S_BASEC_AUX

Address: 0xD83+(C*8)

Access: RW

Reset: 0x0

Figure 33-25 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.

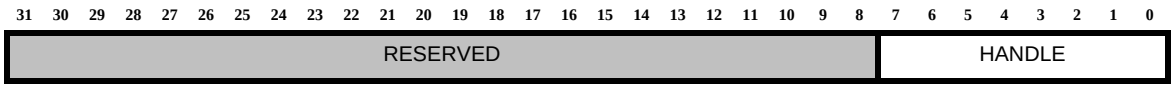
33.4.22 DMA Server Channel Last Register, DMA_S_LASTC_AUX

Address: 0xD84+(C*8)

Access: RW

Reset: 0x0

Figure 33-26 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 is behavior undefined. The same is true if channels with overlapping descriptor ranges are temporarily disabled and reenabled without reset.

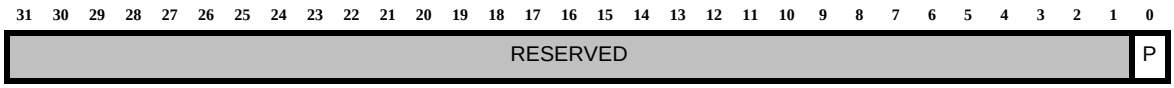
33.4.23 DMA Channel Priority Register, DMA_S_PRIORC_AUX

Address: 0xD85+(C*8)

Access: RW

Reset: 0x0; all channels have same priority

Figure 33-27 DMA_S_PRIORC_AUX Register



The DMA_S_PRIO*C*_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 priority are arbitrated in a round-robin fashion. All channels have equal priorities upon reset.

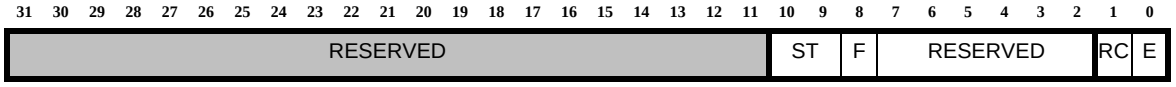
33.4.24 DMA Channel Control Register, DMA_S_STATC_AUX

Address: 0xD86+(C*8)

Access: RW

Reset: 0x0; all channels have same priority

Figure 33-28 DMA_S_STATC_AUX Register



The DMA_S_STAT*C*_AUX register defines the controls and status for channel *C*. The register includes controls for enabling and reseting the channel; the status fields include the channel state and full state.

Table 33-19 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.

33.4.25 CLN_SLV_i_BCR

Build Configuration Registers of CLN slave port connected with DMA server.

When CDMA is configured in the design, one additional slave port is created with *device_id* equal to `CFG_SLV_DEV_COUNT`, that is, `index_of_CDMA_slave_port == CFG_SLV_DEV_COUNT`.

The BCR registers corresponding to this additional internal slave port is present to indicate the hardware details of interface between CDMA and the cluster network.

For more information on these Build Configuration Registers of CLN slave port, see the *ARCv3 ISA Programmer's Reference Manual for ARCv3 Processors*.

33.5 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]                 ; retrun 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, 0xD20   ; 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, 0xD40   ; 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 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".

35 ROM Patching Unit

35.1 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 35-1 illustrates the RPU functionality. Figure 35-2 illustrates the components of the ROM Patching Unit.

Figure 35-1 RPU Functionality

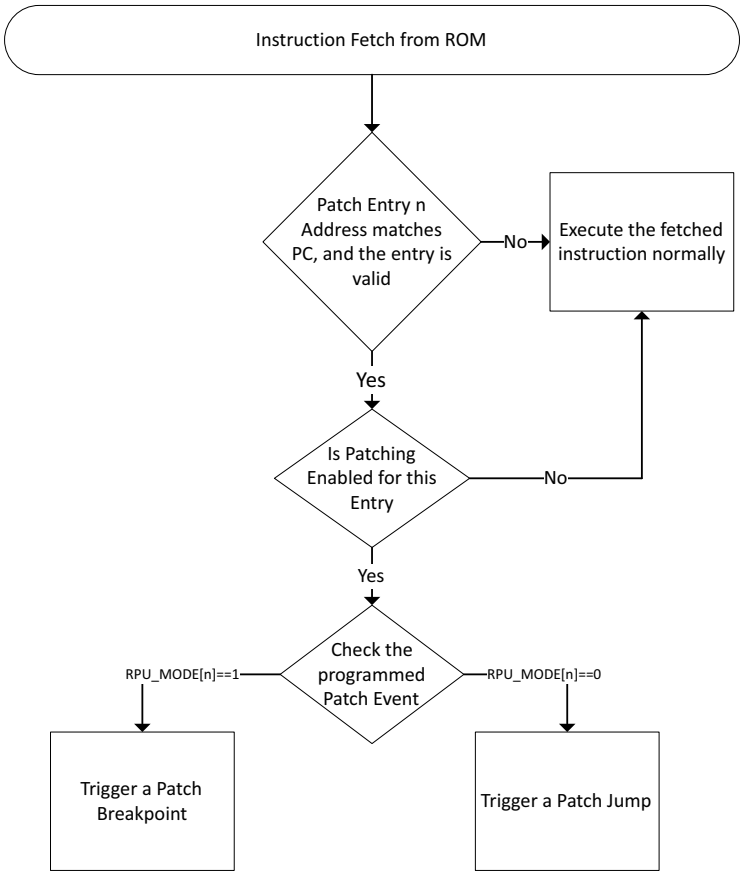
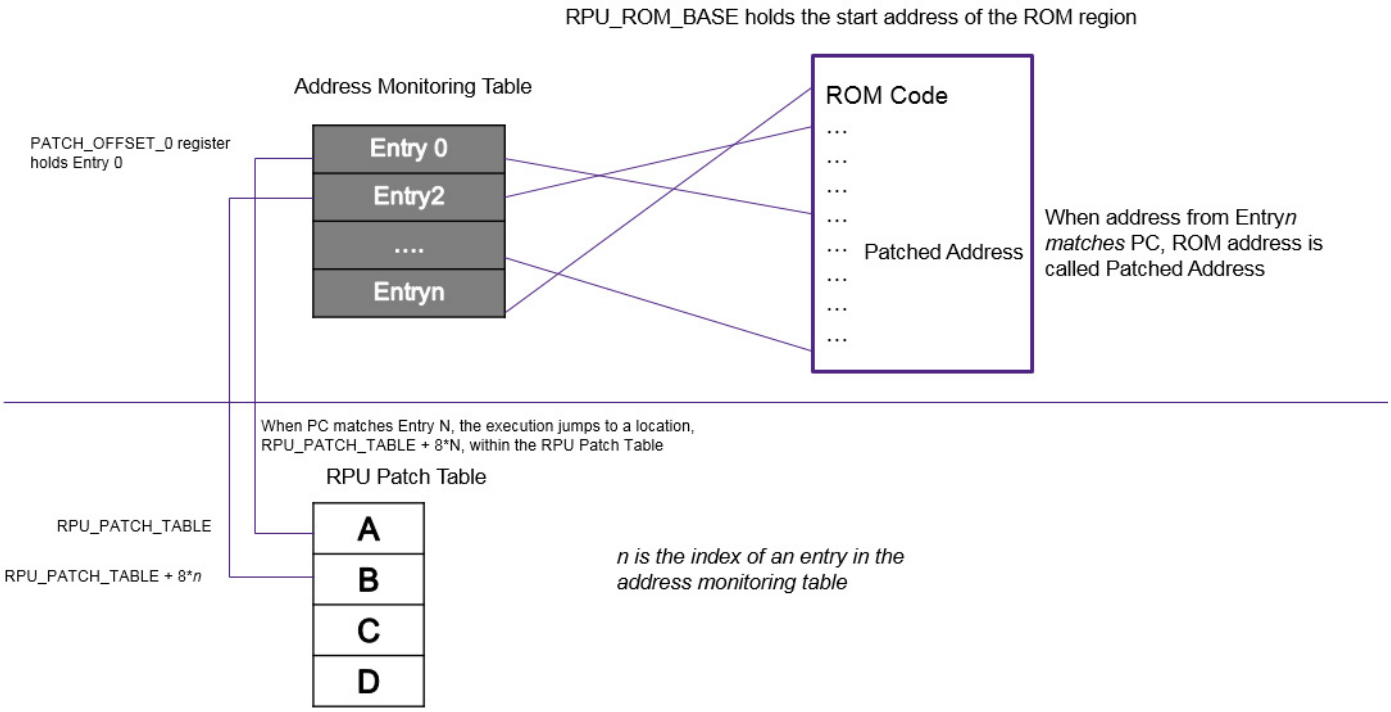


Figure 35-2 RPU Mapping



35.2 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 using the `PatchTable[0].Address` register. 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.

35.3 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.

35.4 Patch Jump

When a Patch Jump is taken, the processor effectively converts the current instruction to an unconditional Jump to the Target address:

```
Target_address = base_of_patch_table + 8 * 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.

35.4.1 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.

35.4.2 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.

35.5 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 entries if the patch addresses do not have 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.

35.6 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`)

35.7 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 the 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.

35.8 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`).

35.9 Auxiliary Registers

The RPU provides an indirect programming interface to program the RPU registers. Table 35-1 lists the indirect programming interface registers. Table 35-3 lists the RPU programming registers.

Table 35-1 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

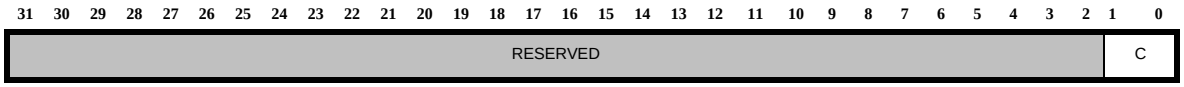
35.9.1 RPU Command Register, RPU_CMND

Address: 0x390

Access: RW

Reset 0x0000_0000

Figure 35-3 RPU_CMND Register



Use this register to perform read or write operations on the RPU programming registers (See Table 35-3). 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_CMND register. The RPU_CMND register is cleared on reset.

Table 35-2 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.

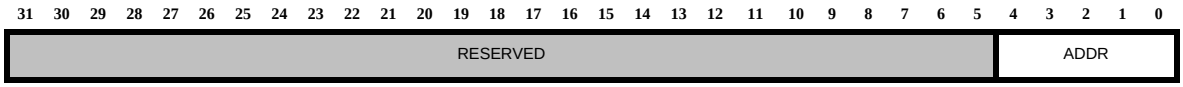
35.9.2 RPU Address Register, RPU_ADDR

Address: 0x391

Access: RW

Reset: 0x0000_0000

Figure 35-4 RPU_ADDR Register



The RPU_ADDR register defines the address of the RPU register that is accessed by an RPU command. See Table 35-3 for a list of valid values for the RPU_ADDR register.

Table 35-3 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 35-3 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

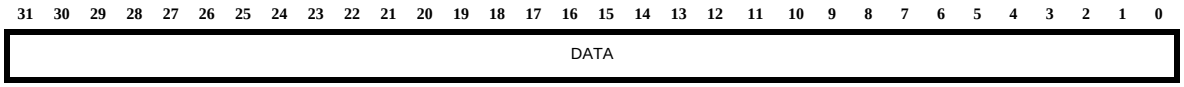
35.9.3 RPU DATA Register, RPU_DATA

Address: 0x392

Access: RW

Reset: 0x0000_0000

Figure 35-5 RPU_DATA Register

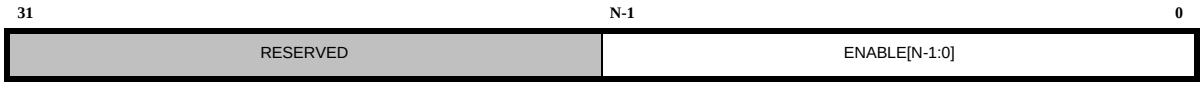


This register contains the data read from the RPU or the data to be written to the RPU. The RPU_ADDR register specifies the RPU address that is being accessed.

35.9.4 RPU Enable Register, RPU_ENABLE

RPU_ADDR value	0x0
Reset	0x0000_0000

Figure 35-6 RPU_ENABLE Register



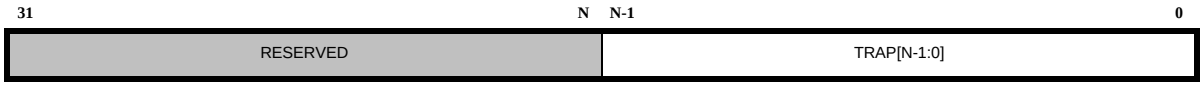
The RPU_ENABLE register implements an individual enable bit for each Patch Entry.

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.

35.9.5 RPU MODE Register, RPU_MODE

RPU_ADDR value	0x1
Reset	0x0000_0000

Figure 35-7 RPU_MODE Register



The RPU_MODE register implements an individual Trap bit for each Patch Entry.

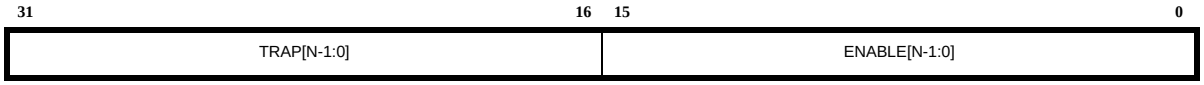
When an enabled Patch Table entry has its Trap bit set to 1 and the processor attempts to execute an instruction from the corresponding address offset within ROM, a Patch Breakpoint is triggered.

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.

35.9.6 RPU Status Register, RPU_STATUS

RPU_ADDR value 0x2

Figure 35-8 RPU_STATUS Register



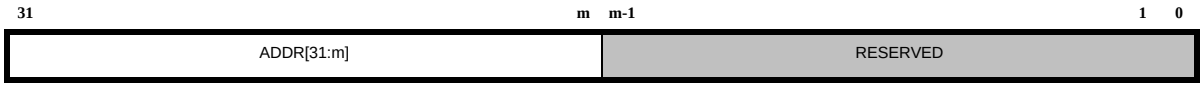
This register indicates the identity of the most recent Patch Entry that has triggered a Patch Breakpoint or a Patch Jump. Each Patch Entry has two bits in this register. For an entry E, bit [E] indicates if the entry has triggered a patch jump. Bit [E+16] indicates if this entry has triggered a breakpoint. 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].

35.9.7 ROM Base Address Register, RPU_ROM_BASE

RPU_ADDR value	0x3
Reset	0x0000_0000

Figure 35-9 RPU_ROM_BASE Register

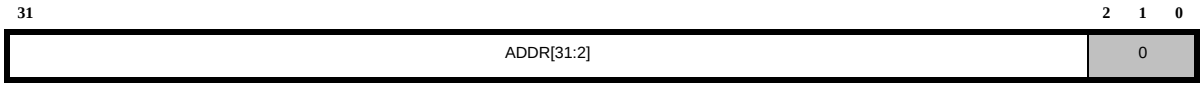


This register holds the size-aligned base address of the ROM. *m* in ADDR[*31:m*]is defined as $\log_2$(ROM_BYTES).

35.9.8 RPU Patch Table Base Address Register, RPU_PATCH_TABLE_BASE

RPU_ADDR value	0x4
Reset	0x0000_0000

Figure 35-10 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.

35.9.9 RPU Patch Entry Register, PatchTable[n].Address

RPU_ADDR value	0x10 to 0x1F
Reset	0x0000_0000

Figure 35-11 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.

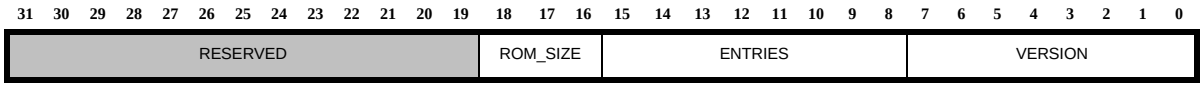
If more than one valid Patch Entry has the same address field, multiple entries can be triggered simultaneously. This is a software error, and RPU behavior is undefined in this case.

35.9.10 RPU Build Configuration Register, RPU_BUILD

Address: 0xFC

Access: R

Figure 35-12 RPU_BUILD Register



This register specifies the RPU version and its build-time configuration.

Table 35-4 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 `-rpu_patch_entries` configuration option. When `-rpu_patch_entries = 0`, 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 `-rpu_rom_size` configuration option.

Part 3
FastMath Extension Pack

In this part:

- FastMath Data Organization and Addressing
- FastMath Register Set Extensions
- FastMath Extension Instruction Details

36 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

36.1 FastMath Data Organization and Addressing

This section describes the data formats used by the ARCv3 ISA.

36.1.1 Fractional Data Formats

The set of extension instructions enabled by the ARCv3 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 ARCv3 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:

$$\texttt{NQM\_MIN} = \frac{-2^{k-1}}{2^{m}} = ULP(-2^{k-1})$$

to:

$$\texttt{NQM\_MAX} = \frac{2^{k-1}-1}{2^{m}} = \mathrm{ULP}(2^{k-1}-1)$$

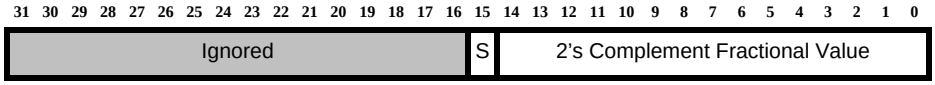
Thus, the mathematical value of the 2's complement integer I, encoded in an nQm format is simply $\frac{I}{2^{m}}$.

36.1.2 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.

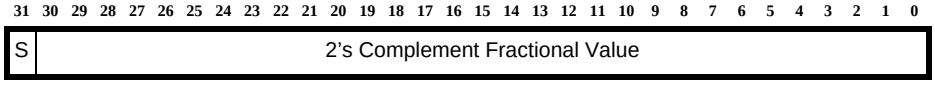
Figure 36-1 16-Bit Signed 1Q15 Data Representation in a 32-bit Register



36.1.3 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.

Figure 36-2 32-Bit Signed Fractional Data



36.2 FastMath Register Set Extensions

36.2.1 Extension Auxiliary Registers

Table 36-1 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 36-1 FMP Auxiliary Registers

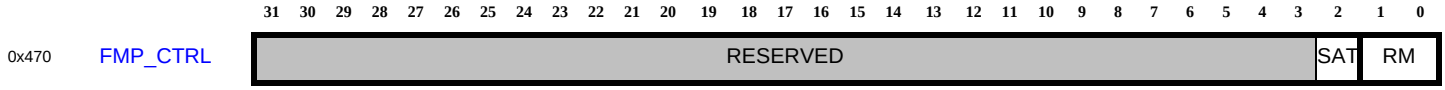
Address	Auxiliary Register Name	Access	Description
0x470	FMP Control Register, FMP_CTRL	rw	Register to control the rounding and guard bit behavior.

36.2.2 FMP Control Register, FMP_CTRL

Address: 0x470

Access: rw

Figure 36-3 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 36-2 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.

36.2.3 Build Configuration Registers

Table 36-3 Build Configuration Registers

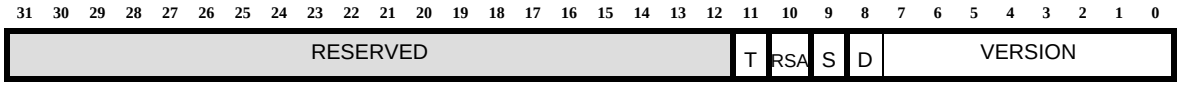
Number	Name	Access	Description
0xE8	FMP Build Configuration Register, FMP_BUILD	R	Build configuration for FMP

36.2.3.1 FMP Build Configuration Register, FMP_BUILD

Address: 0xE8

Access: R

Figure 36-4 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 36-4 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).

36.2.4 Extension Core Registers

The FastMath extension pack does not define any extension core registers.

36.3 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 ARCv3 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 ARCv3 instructions, other customer-defined instructions.

All FastMath Pack instruction encodings and extension register addresses are distinct from existing ARCv3 encodings and addresses.

36.3.1 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.

36.3.2 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;
```

36.3.3 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; }
```

36.3.4 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;
   }
}
```

36.3.5 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));
}
```

36.3.6 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(0x0000000100000000),
    sint64(0xFFFFFFFB10B0CD92),
    sint64(0x0000000103C1F06C),
    sint64(0xFFFFFFFFEAA2C3FC),
    sint64(0x0000000000F0F926),
    sint64(0xFFFFFFFFFFF966BA),
    sint64(0x0000000000001E26),
  };

   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;
}
```

36.3.7 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;
}
```

36.3.8 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) 0.888888888888889 )
   sint64(0x0000000666666666), // FLT_2_Q31((double) 0.800000000000000 )
   sint64(0x00000005d1745d17), // FLT_2_Q31((double) 0.727272727272727 )
   sint64(0x0000000555555555), // FLT_2_Q31((double) 0.666666666666667 )
   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 $\log_2$_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(0x0000000000000010)  // FLT_2_Q31((double) -0.000000000000000 )
};
```

The Taylor expansion coefficients are nominally defined such that each coefficient i is given by: $\frac{(-1)^{(i+1)}}{2i\ln 2}$

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(0xfffffffd1d5ba202), // FLT_to_2Q35((double) -0.360665067997034 )
      sint64(0x00000001ed25e531), // FLT_to_2Q35((double)  0.240794935777816 )
      sint64(0xfffffffe9b0ae48b), // 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);
}
```

36.3.9 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) 1.000000000000000) = 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);
}
```

36.3.10 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;
}
```

36.3.11 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;
}
```

36.3.12 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(0x0000000145F306CC),
      sint64(0xFFFFFFFF9359AE62),
      sint64(0x00000000412FF174),
      sint64(0xFFFFFFFFD17846C4),
      sint64(0x0000000023F6DCFA),
      sint64(0xFFFFFFFFE389D938),
      sint64(0x000000001573A14D),
      sint64(0xFFFFFFFFF219CCC1),
      sint64(0x0000000006D307B0),
      sint64(0xFFFFFFFFFDD7F5B6),
      sint64(0x000000000051F0F6)
   };

   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);
}
```

36.3.13 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));
}
```

36.3.14 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 = 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	00111bbb00100010FBBBCCCCCCAAAAAA
FMP_ADDS<.f>	a,b,u6	00111bbb01100010FBBBuuuuuuAAAAAA
FMP_ADDS<.f>	b,b,s12	00111bbb10100010FBBBssssssSSSSSS
FMP_ADDS<.cc><.f>	b,b,c	00111bbb11100010FBBBCCCCCC0QQQQQ
FMP_ADDS<.cc><.f>	b,b,u6	00111bbb11100010FBBBuuuuuu1QQQQQ

FMP_RNDH

Function

Round and saturate a 32-bit signed integer to a 16-bit signed fraction

Extension Group

APEX FastMath Extensions (`-support_sat==true`)

Operation

```
b = 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	00111bbb00101111FBBBCCCCCC101001
FMP_RNDH<.f>	b,u6	00111bbb01101111FBBBuuuuuu101001

FMP_SATH

Function

Saturate a 32-bit value to a 16-bit fractional value

Extension Group

APEX FastMath Extensions (`-support_sat==true`)

Operation

```
b = 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	00111bbb00101111FBBBCCCCCC101000
FMP_SATH<.f>	b,u6	00111bbb01101111FBBBuuuuuu101000

FMP_DIVF

Function

Signed 32-bit fractional division.

Extension Group

APEX FastMath Extensions (`-support_div==true`)

Operation

```
a = (b << 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 abs(B) <= 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.

Pseudo Code

```
if (cc == true) {                                                    /* FMP_DIVF */
   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	00111bbb00100000FBBBCCCCCCAAAAAA
FMP_DIVF<.f>	a,b,u6	00111bbb01100000FBBBuuuuuuAAAAAA
FMP_DIVF<.f>	b,b,s12	00111bbb10100000FBBBssssssSSSSSS
FMP_DIVF<.cc><.f>	b,b,c	00111bbb11100000FBBBCCCCCC0QQQQQ
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 << 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 abs(B) <= 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.

Pseudo Code

```
if (cc == true) {                                                    /* FMP_DIVF15 */
   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	00111bbb00100001FBBBCCCCCCAAAAAA
FMP_DIVF15<.f>	a,b,u6	00111bbb01100001FBBBuuuuuuAAAAAA
FMP_DIVF15<.f>	b,b,s12	00111bbb10100001FBBBssssssSSSSSS
FMP_DIVF15<.cc><.f>	b,b,c	00111bbb11100001FBBBCCCCCC0QQQQQ
FMP_DIVF15<.cc><.f>	b,b,u6	00111bbb11100001FBBBuuuuuu1QQQQQ

FMP_RECIP

Function

Signed 32-bit reciprocal.

Extension Group

APEX FastMath Extensions (`-support_div==true`)

Operation

```
b = (1 << 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.

Pseudo Code

```
if (c != 0) {                                                    /* FMP_RECIP */
   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;
   }
}
```

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	00111bbb00101111FBBBCCCCCC101010
FMP_RECIP<.f>	b,u6	00111bbb01101111FBBBuuuuuu101010

FMP_RECIP15

Function

Signed 16-bit reciprocal.

Extension Group

APEX FastMath Extensions (`-support_div==true`)

Operation

```
b= (1 << 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.

Pseudo Code

```
if (c != 0) {                                                    /* FMP_RECIP15 */
   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	00111bbb00101111FBBBCCCCCC101011
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.

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	00111bbb00101111FBBBCCCCCC100000
FMP_SQRTF<.f>	b,u6	00111bbb01101111FBBBuuuuuu100000

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.

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	00111bbb00101111FBBBCCCCCC100001
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.

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	00111bbb00101111FBBBCCCCCC011110
FMP_COS<.f>	b,u6	00111bbb01101111FBBBuuuuuu011110

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.

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	00111bbb00101111FBBBCCCCCC101100
FMP_COS15<.f>	b,u6	00111bbb01101111FBBBuuuuuu101100

FMP_SIN

Function

Compute the sine of a Q31 fractional operand

Extension Group

APEX FastMath Extensions (`-support_trig==true`)

Operation

```
b = sine_q31(c);
```

Instruction Format

op b,c

Syntax Example

FMP_SIN<.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 Q31 fractional input operand C. The input operand represents an angle expressed in radians divided by Pi. The resulting signed Q31 sine value is returned to the destination operand B.

Pseudo Code

```
b = fmp_sine_q31(c);                                          /* FMP_SIN */
if (F) {
   STATUS32.Z = b == 0x0000_0000;
   STATUS32.N = (b < 0);
}
```

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	00111bbb00101111FBBBCCCCCC011111
FMP_SIN<.f>	b,u6	00111bbb01101111FBBBuuuuuu011111

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.

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	00111bbb00101111FBBBCCCCCC101101
FMP_SIN15<.f>	b,u6	00111bbb01101111FBBBuuuuuu101101

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.

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	00111bbb00101111FBBBCCCCCC100101
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.

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	00111bbb00101111FBBBCCCCCC101110
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	•	= N must be cleared, as the result of 2^x can never be < 0
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 cleared.

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	00111bbb00101111FBBBCCCCCC100111
FMP_EXP2<.f>	b,u6	00111bbb01101111FBBBuuuuuu100111

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	•	= N flag should be cleared
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 cleared.

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	00111bbb00101111FBBBCCCCCC101111
FMP_EXP215<.f>	b,u6	00111bbb01101111FBBBuuuuuu101111

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.

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	00111bbb00101111FBBBCCCCCC100110
FMP_LOG2<.f>	b,u6	00111bbb01101111FBBBuuuuuu100110

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.

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	00111bbb00101111FBBBCCCCCC110000
FMP_LOG215<.f>	b,u6	00111bbb01101111FBBBuuuuuu110000

Part 4

ARCv3 Floating-Point Unit

In this part:

- Floating-Point Unit
- Floating-Point Auxiliary Registers
- Floating-Point Unit Instructions

37 Floating-Point Unit

37.1 Key features of the ARCv3 Floating-point ISA

The ARCv3 floating-point ISA supports the following features:

- Functionally compatible with current and future, scalar and vector, FPU implementations in ARC HS and DW EV products.
- A separate set of (up to) 32 floating-point registers, `f0-f31`, each up to 64 bits wide. The actual number is configurable, and may be 8, 16, or 32.
- A set of floating-point load/store instructions for moving values between memory and the floating-point registers.
- A set of floating-point instructions, supporting all mandatory IEEE 754-2008 operators, and providing features at least as rich as those found in the RISC-V 'F' and 'D' extensions.Supports optional half-precision (16-bit) floating-point types.
- Supports optional vector floating-point, including 2 x single-precision, and 4 x half-precision operations.
- Optional double-width vector capabilities, including 2 x double-precision, 4 x single-precision, and 8 x half-precision. The operands of these double-width operations are defined as even-numbered pairs of floating-point registers.
- Although the ARCv3 instruction set allows a subset of floating-point registers to be configured, the ARCv3 core always configures the full set of 32 floating-point registers.

37.2 Floating-Point Registers

The ARCv3 ISA introduces up to 32 new floating-point registers when `-has_fp` is true. These are designated f0 to f31. The size of each floating-point register (`fpr_width`) can be either 32 or 64 bits, depending on the configuration. `fpr_width` is 64 whenever double-precision operations are configured (`-fp_dp_option == true`) and 32 whenever double-precision operations are de-configured (`-fp_dp_option == false`). The number of floating-point registers is configured with the `-fp_num_regs` option, and can be 8, 16 or 32.

When the `-fp_num_regs` option is 8, registers f0 – f7 are configured.

When the `-fp_num_regs` option is 16, registers f0 – f15 are configured.

If a floating-point instruction produces a result that is narrower than a floating-point register, the result is aligned to the least-significant end of the register and any register bits beyond the most-significant bit of the result are cleared.

For information on how the ARCv3 ABI defines the usage for floating point registers (FPRs), see "Core Register Usage in the ABI".

37.3 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 1are in the following combinations:
> `0 * infinity`
> `or`
> `infinity * 0`
> the default NaN is returned and the instruction raises an invalid flag.

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

37.4 Floating-Point Instruction Encodings

The major opcode **0x1C** defines the instruction as a 32-bit format: **F32_FP_OPS**.

The layout of this format, and its sub-formats, is shown in Table 37-1. A dash (-) indicates an unused field, which must be set to 0 by the assembler and which raises an Illegal Instruction exception if non-zero. Unused fields contain dashes in this table to indicate the spare encoding capacity in this format.

Table 37-1 The F32_FP_OPS Instruction Formats

Format	Sub-format		[31:27]	[26:24]	23	22	21	20	19	18	17	16	15	14	13	12	11	[10:6]	5	4	3	2	1	0
F32_FP_OPS	FP_TOP	>	0x1C	fs2[2:0]	fs3					topf[3:1]			P		fs2[4:3]		1	fd	topf[0]	fs1				
	FP_DOP	>			0	0	-	dopf									0	fd	1	fs1				
	FP_CVF2F	>			0	1	1	0	-	-	cvtf		-				0	fd	1	u4	u3	1	0	u0
	FP_RND	>			0	1	1	0	-	-							0	fd	1	0	u3	1	1	0
	FP_CVF2I	>			0	1	1	0	-	-							0	rd	1	0	u3	0	u1	1
	FMVF2I	=			0	1	1	0	-	-							0	rd	1	1	0	0	0	1
	FP_SOP	>		-	0	1	0	0	0	-	sopf		P		-		0	fd	1	fs1				
	FP_COP	>		Q[2:0]	0	1	0	0	1	-	copf		P		Q[4:3]		0	fd	1	fs1				
	FP_ZOP	>		-	0	1	1	1	-	-	zopf		-		-		0	-	1	-				
	FP_VMVI	>		u[2:0]	0	1	0	1	-	-	vmvf		P		u[4:3]		0	fd	1	fs1				
	FP_VMVR	>		B[2:0]	1	0	-	-	-	-			P		B[4:3]		0	fd	1	fs1				
	FP_CVI2F	>			1	1	1	0	-	-	cvtf		-		B[4:3]		0	fd	1	0	u3	0	u1	0
	FMVI2F	=			1	1	1	0	-	-			-				0	fd	1	1	0	0	0	0
	RESERVED																0		0					

Table 37-2 The F32_FP_MEM Instruction Formats

Format	Sub-format		[31:27]	[26:24]	23	22	21	20	19	18	17	16	15	14	13	12	11	[10:6]	5	4	3	2	1	0
F32_FP_MEM	FP_LOAD	>	0x0D	B[2:0]	S[7:0]								S8	B[5:3]			0	fd	d	a a		ZZ		0
	FP_STORE	>		B[2:0]									S8	B[5:3]			0	fs1	d	a a		ZZ		1

All sub-formats specify a 5-bit register in bits [10:6] and for all instructions except stores, this register is a destination register. The destination register is normally a floating-point register, but can be an integer register in the case of conversion operators (in the FP_CVF2I format) and the move to integer register

instruction (FMVF2I). For store instructions, bits [10:6] specify the floating-point store data register (which is actually a source operand).

The fs1, fs2, and fs3 operands always specify a floating-point source register. The B operand always specifies a general-purpose source register. Some formats contain unsigned literal fields (U5 or u[4:3] and u[2:0], or individual bits u4, u3, u2, u1 and u0). These may be either an index into a vector (in the FP_VMVI format) or format conversion function bits (in the FP_CVF2F, FP_CVF2I and FP_CVI2F formats).

Opcode bit 11 distinguishes between triple-operand floating-point operations (FP_TOP format) and all other FP operations. Bit 5 distinguishes between load/store operations (0) and the remaining floating-point arithmetic operations (1). Within the load/store operations bit 0 distinguishes between loads (0) and stores (1). Values 00, 01 and 10 encode the dual-operand, single-operand and conversion sub-formats in opcode bits [23:22].

The sub-formats of F32_FP_OPS are encoded to simplify the identification of B as a source. This is because identification of integer source registers is a time-critial decoding task in most implementations. This format reads a B-register if bit[11] == 0 and either bit[23] == 1 or bit[5] == 0.

The two formats FP_VMVI and FP_VMVR encode instructions to move scalar FP values into or out of a vector of FP values. The "I" form uses a literal index (U5) to select a position within a vector, whereas the "R" form uses a general-purpose register (B) to specify a vector index. Instructions in these formats are legal only when `-fp_vec_option == true`.

The FP_CVF2F format encodes instructions that convert from one floating-point type to another. The FP_CVF2I format encodes instructions that convert from floating-point types to integer types; whereas the FP_CVI2F format encodes instructions that convert in the opposite direction. There are two formats that each encode single instruction; FMVI2F moves the contents of an integer register to a floating-point register, whereas FMVF2I moves values in the other direction. Neither of these two instructions performs any format conversions.

The FP_TOP, FP_DOP, FP_SOP, FP_VMVI and FP_VMVR formats all have a common "operator precision" operand (P) in bits [15:14], to indicate if their precision is Half (00), Single (01) or Double (10). The value P = 11 is reserved and is an illegal value in these three sub-formats. An Illegal Instruction exception is also raised on any attempt to execute a floating-point instruction that specifies an unsupported value of P. This occurs, when P = 00 and `-fp_hp_option` is `false`, or if P = 10 and `-fp_dp_option` is `false`.

The precision, syntax and legality of all P values is shown in Table 37-3.

Table 37-3 Interpretation and Legality of <P> Field

Precision	Syntax	P	-fp_hp_option		-fp_dp_option	
			false	**true**	**false**	**true**
Half	"H"	0 0	Illegal	√	N/A	N/A
			Illegal	√	N/A	N/A
			Illegal	√	N/A	N/A
			Illegal	√	N/A	N/A

Table 37-3 Interpretation and Legality of <P> Field

Precision	Syntax	P	-fp_hp_option		-fp_dp_option	
Single	"S"	0 1	√	√	√	√
			√	√	√	√
			√	√	√	√
			√	√	√	√
Double	"D"	1 0	N/A	N/A	Illegal	√
			N/A	N/A	Illegal	√
			N/A	N/A	Illegal	√
			N/A	N/A	Illegal	√
Undefined	-	1 1	Illegal	Illegal	Illegal	Illegal

37.4.1 Floating-point Load/Store Instruction Formats

The load/store formats each specify a 6-bit integer base register *B*, a 9-bit signed offset *S*, and the standard memory operation modifiers <aa> and <ZZ>.

The *B*, *S*, <aa> and <ZZ> fields of the FP_LOAD and FP_STORE sub-formats are in identical positions to those same fields in the F32_ST_OFFSET format (0x03). Further, the B and S fields are also in the same positions as in the F32_LD_OFFSET format (0x02). This simplifies the decoding of these formats, which is particularly relevant for the integer base register B and the signed offset S.

The encoding of floating-point load/store instructions using the <ZZ> field is illustrated below in Table 37-4, and this also indicates instructions that are illegal in certain combinations of core and floating-point unit configuration. The **8B** and **16B** columns indicate whether the core configuration supports load and store operations that transfer 8 bytes and 16 bytes per instruction respectively ("X" indicates cases where these capabilities are not relevant).

Table 37-4 Interpretation of <ZZ> bits in FP Load/Store Instructions

<ZZ>	Size (bits)	<d>	8B	16B	FPR_WIDTH = 32		FPR_WIDTH = 64	
					FP load insts	FP load insts	FP load insts	FP load insts
10	16	0	X	X	FLD16	FST16	FLD16	FST16
00	32	0	X	X	FLD32	FST32	FLD32	FST32
		1	0	X	Illegal	Illegal	n/a	n/a
			1	X	FLDD32	FSTD32	FLDD32	FSTD32
01	64	0	0	X	Illegal	Illegal	n/a	n/a
			1	X			FLD64	FST64

Table 37-4 Interpretation of <ZZ> bits in FP Load/Store Instructions

<ZZ>	Size (bits)	<d>	8B	16B	FPR_WIDTH = 32		FPR_WIDTH = 64	
					FP load insts	FP load insts	FP load insts	FP load insts
01	64	1	X	0	Illegal	Illegal	Illegal	Illegal
			X	1			FLDD64	FSTD64

Example syntax: `FLD32.A F4, [R0, 16]`

The minimum core configurations required to support 8B and 16B data transfers are defined below in Table 37-5.

Table 37-5 Supported Memory Data Transfer Sizes as a Function of Core Configuration

Memory Data Transfer Size	Required Core Configuration
8B (8 bytes)	ARC64 **or** `–ll64_option` **or** `–fp_dp_option`
load 16B (16 bytes)	`(–m128_option)` **and** `–fp_dp_option`
store 16B (16 bytes)	`–m128_option` **and** `–fp_dp_option`

Transfers of up to 8 bytes are always supported by an ARC64 core, or . Further, if an ARC32 core has double-precision floating-point configured, this forces the core to support 8-byte memory operations .

Transfers of up to 16 bytes are supported only by ARC64 cores, and only under following conditions:

- Floating-point stores of 128 bits are supported only when both the -m128_option and double-precision floating-point are configured.
- Floating-point loads of 128 bits are supported when double-precision floating-point is configured and -m128_option is configured.

The number of bits transferred between FPRs and memory is always either `FPR_WIDTH` or `2*FPR_WIDTH`. When it is `2*FPR_WIDTH`, the FP load/store instructions transfer an even-numbered pair of FP registers. If an odd-numbered register pair is specified, then an Illegal Instruction exception is raised.

The scaling shifts applied to the offset of a floating-point load/store instruction when using the `.AS` addressing mode are defined according to the <ZZ> field as shown in Table 37-6.

Table 37-6 Supported Memory Data Transfer Sizes as a Function of Core Configuration

<ZZ>	Scaling shift (.AS addressing mode only)
10	1
00	2
01	3
11	3

The `FLD16` instruction loads a 16-bit value from the specified memory address and places it into the lower 16 bits of the destination FP register. All bits above bit 15 in the destination FP register are cleared.

The FLD32 instruction loads a 32-bit value from the specified memory address and places it into the lower 32 bits of the destination FP register. If double-precision FP is configured, then all bits above bit 31 in the destination FP register are cleared.

The FLD64 instruction loads a 64-bit value from the specified memory address. If FPR_WIDTH=64, the loaded value is placed in the specified destination FP register. If FPR_WIDTH=32, the lower 32 bits of the loaded value are placed in the specified FP register and the upper 32 bits are placed in the next FP register, that is the result is written to a pair of single-precision FP registers. In common, with all register-pair operands, the specified register must be even-numbered.

The FLD128 instruction loads two consecutive 64-bit values from the specified memory address and places element 0 in the named destination FP register. Element 1 is placed in the next sequential FP register.

The FST16 instruction stores a 16-bit value from the lower 16 bits of the designated FP source register to memory, at the specified address.

The FST32 instruction stores a 32-bit value from the lower 32 bits of the designated FP source register to memory, at the specified address.

The FST64 instruction stores a 64-bit value from one double-precision FP source register, or an even-numbered pair to single-precision FP source registers, to memory at the specified address.

The FST128 instruction stores two 64-bit values from an even-numbered pair of double-precision FP source registers, to memory at the specified address. The even-numbered FP register is stored to the lower 64-bit address and the odd-numbered FP register is stored to the higher 64-bit address.

37.4.2 Floating-point Arithmetic Operation Formats

Within the arithmetic operations there are 3 sub-formats; the triple-operand format (FP_TOP), the dual-operand format (FP_DOP), and the single-operand format (FP_SOP). Floating-point mnemonics containing the string "<P>" are replicated for all legal values of *P*. Thus, "F<P>ADD" defines instructions FSADD, FDADD (as in the ARCv2 ISA), as well as FHADD (when the `-fp_hp_option` is `true`).

The FP_TOP, FP_DOP, FP_SOP and FP_COP formats all have a 5-bit floating-point source register field fs1 in bit range [4:0]. The FP_TOP and FP_DOP formats also share a common floating-point source register field fs2 (split into 2 regions within the instruction word, in bit-ranges [13:12] and [26:24]). In addition, the FP_TOP format has a third floating-point source register field fs3 in bits [23:19].

Note All floating-point register operand fields specify a floating-point register; there is no way to select a LIMM value as a literal operand for a floating-point instruction. Floating-point constants must be loaded into an FP register.

The FP_TOP format has a 3-bit function field (topf) in bits [18:16]. The FP_DOP format has a 5-bit function field (dopf) in bits [20:16], the FP_SOP, FP_COP and FP_ZOP formats each have a 2-bit function field (sopf, copf and zopf respectively) in bits [17:16].

The FP_COP format has a 5-bit condition code field (Q), with Q[4:3] in bits [13:12] and Q[2:0] in bits [26:24]. This field selects a baseline ARC condition code in exactly the same way as for integer instructions. Note: this field does not permit APEX extension condition codes to be encoded.

All arithmetic functions that round their result use the rounding mode specified in the Rounding Mode (RM) field of the floating-point control register (FP_CTRL,at auxiliary address 0x300).

The triple-operand floating-point instructions supported in ARCv3 are listed in Table 37-7. These are each variants of a fused multiply-add operation, and they all follow the IEEE 754-2008 standard [9] in it's definition of the behavior of fusedMultiplyAdd operations. ARCv3 supports four variants of fusedMultiplyAdd operation which vary in how they selectively negate one or two input operands prior to performing the canonical fusedMultiplyAdd operation (a×b)+c. The rules for deriving a, b, and c, from instruction operands s1, s2 and s3, are listed in Table 37-7.

Note As the IEEE multiply operation (a×b) defines the sign of it's product as the XOR of the signs of a and b, the MSUB and NMADD instruction classes can negate either s1 or s2 with equivalent results.

The triple-operand floating-point instructions supported in ARCv3 are listed in Table 37-8. This includes scalar and vector versions of each supported fused multiply-add operation, as well as a version of the vector instructions in which the s2 operand is scalar quantity that is replicated in each elemental computation within the vector instruction.

Table 37-7 Negation of operands for fusedMultiplyAdd operations

Instruction Class	Operation	a	b	c
MADD	fd = s3 + (s1 * s2)	s1	s2	s3
MSUB	fd = s3 - (s1 * s2)	negate(s1)	s2	s3
NMADD	fd = -(s3 + (s1 * s2))	negate(s1)	s2	negate(s3)
NMSUB	fd = -(s3 - (s1 * s2))	s1	s2	negate(s3)

When `-fp_vec_option` is true, the vector versions of the floating-point add/sub/mul, and all fused versions of those operations, are supported for each operator precision that is configured. For all vector instructions, opcode bit topf[3] indicates if the second floating-point source operand register (s2) of the multiply operation is a vector (0) or a scalar (1). All scalar s2 operands have precision given by the P field and their least-significant bit is aligned to bit 0 of the source register.

Table 37-8 Triple-operand Floating-point Instructions

Instruction	topf [3:0]	Operation
F<P>MADD	0x0	fd = s3 + (s1 * s2)
F<P>MSUB	0x1	fd = s3 - (s1 * s2)
F<P>NMADD	0x2	fd = - (s3 + (s1 * s2))
F<P>NMSUB	0x3	fd = - (s3 - (s1 * s2))
VF<P>MADD	0x4	vectorized F<P>MADD vector + (vector x vector)
VF<P>MSUB	0x5	vectorized F<P>MSUB vector – (vector x vector)
VF<P>NMADD	0x6	vectorized F<P>NMADD – (vector + (vector x vector))
VF<P>NMSUB	0x7	vectorized F<P>NMSUB – (vector – (vector x vector))

Table 37-8 Triple-operand Floating-point Instructions

Instruction	topf [3:0]	Operation
-	0x8 – 0xB	(unused and illegal)
VF<P>MADDS	0xC	vectorized F<P>MADD vector + (vector x scalar)
VF<P>MSUBS	0xD	vectorized F<P>MSUB vector – (vector x scalar)
VF<P>NMADDS	0xE	vectorized F<P>NMADD – (vector + (vector x scalar))
VF<P>NMSUBS	0xF	vectorized F<P>NMSUB – (vector – (vector x scalar))

When `-fp_wide_option` is `true`, each vector operand is twice the width of the largest supported floating-point type. The width of a vector floating-point operation is given by `VFP_WIDTH`. The influence of FPU configuration options on `FPR_WIDTH` and `VFP_WIDTH` are defined in Table 37-9.

Table 37-9 FPR_WIDTH and VFP_WIDTH Definitions

-fp_dp_option	-fp_wide_option	FPR_WIDTH	VFP_WIDTH	FPRs per Vector Operand
false	false	32	32	1
false	true	32	64	2
true	false	64	64	1
true	true	64	128	2

When the number of FPRs per vector operand is 2, then each such operand must be specified by an even-numbered FP register specifier. Specifying an odd-numbered FP register in such cases will raise an Illegal Instruction exception. When `-fp_wide_option` is `false`, each vector operand is a single floating-point register.

The number of elements in each short vector is given by `VFP_WIDTH/B`, where `B` is the intrinsic bit-width of the floating point precision defined by the P field (half: B=16, single: B=32, double: B=64). Thus, if `VFP_WIDTH = 128`, then the `VFHMADD` instruction will perform a fused multiply-add operation on 128-bit operands, each containing a vector of 8 half-precision elements.

In the minimal vector FP configuration, when `VFP_WIDTH = 32`, the `VFSMADD` instruction operates on vectors containing just one single-precision elements.

The maximum width of floating-point vector operation supported by each processor configuration is denoted `VFP_MAX_VLEN`, the values of which are enumerated as a function of the floating-point configuration in Table 37-10.

Table 37-10 Enumerated values of VFP_MAX_VLEN as a function of configuration

-fp_dp_option	-fp_wide_option	-fp_hp_option	VFP_MAX_VLEN

Table 37-10 Enumerated values of VFP_MAX_VLEN as a function of configuration

false	false	false	1
		true	2
	true	false	2
		true	4
true	false	false	2
		true	4
	true	false	4
		true	8

The dual-operand floating-point instructions supported are listed in Table 37-11. These include: the add/sub/mul/div/min/max operations; operations to manipulate the sign of a floating-point value; floating-point relational tests; and vectorized versions of add/sub/mul operations (available when the option `-fp_vec_option` is `true`).

The three sign-manipulation functions (F<P>SGNJ, F<P>SGNJN, F<P>SGNJX). Floating-point ABS, NEG, and MOV can be implemented using these sign-manipulation instructions as follows:

```
F<P>ABS f1, f2 : F<P>SGNJX f1, f2, f2
F<P>NEG f1, f2 : F<P>SGNJN f1, f2, f2
F<P>MOV f1, f2 : F<P>SGNJ f1, f2, f2
```

Table 37-11 Dual-operand Floating-point Instructions

Instruction	dopf [4:0]	Operation
F<P>ADD	0x00	fd = s1 + s2
F<P>SUB	0x01	fd = s1 – s2
F<P>MUL	0x02	fd = s1 * s2
F<P>DIV	0x03	fd = s1 / s2
F<P>CMP	0x04	ZNCV ← F<P>CMP(s1, s2) "quiet" floating-point comparison
F<P>CMPF	0x05	ZNCV ← F<P>CMP(s1, s2) "signaling" floating-point comparison
F<P>MIN	0x06	fd = (s1 > s2) ? s2 : s1
F<P>MAX	0x07	fd = (s1 > s2) ? s1 : s2
F<P>SGNJ	0x08	fd.{s, e, m} = {s2.s, s1.e, s1.m}
F<P>SGNJN	0x0A	fd.{s, e, m} = {not(s2.s), s1.e, s1.m}
F<P>SGNJX	0x0B	fd.{s, e, m} = {xor(s1.s, s2.s), s1.e, s1.m}
-	0x0C-0x0F	(unused and illegal)
VF<P>ADD	0x10	vectorized F<P>ADD (vector + vector)
VF<P>SUB	0x11	vectorized F<P>SUB (vector – vector)
VF<P>MUL	0x12	vectorized F<P>MUL (vector * vector)

Table 37-11 Dual-operand Floating-point Instructions (Continued)

Instruction	dopf [4:0]	Operation
VF<P>DIV	0x13	Vectorized F<P>DIV (vector / vector)
VF<P>ADDS	0x14	vectorized F<P>ADD (vector + scalar)
VF<P>SUBS	0x15	vectorized F<P>SUB (vector – scalar)
VF<P>MULS	0x16	vectorized <P>FMUL (vector * scalar)
VF<P>DIVS	0x17	Vectorized F<P>DIV (vector / scalar)
VF<P>UNPKL	0x18	vector unpack, least-significant half result
VF<P>UNPKM	0x19	vector unpack, most-significant half result
VF<P>PACKL	0x1A	vector pack, least-significant half result
VF<P>PACKM	0x1B	vector pack, most-significant half result
VF<P>BFLYL	0x1C	vector butterfly, least-significant half result
VF<P>BFLYM	0x1D	vector butterfly, most-significant half result
VF<P>ADDSUB	0x1E	vectorized alternating F<P>ADD/F<P>SUB (vector +- vector)
VF<P>SUBADD	0x1F	vectorized alternating F<P>SUB/F<P>ADD (vector -+ vector)

The dual-operand vector floating-point operations are included only when `-fp_vec_option = true`. When this option is `false`, any attempt to execute vector floating-point operations will raise an Illegal Instruction exception.

Each of the dual-operand vector floating-point functions (add, sub, mul) have two versions; one that takes {vector x vector} operands and one that {vector x scalar} operands.

The VF<P>ADDSUB and VF<P>SUBADD instructions implement additions and subtractions in alternate lanes of the vector operation. The ADDSUB instruction performs addition in all even-numbered lanes and subtraction in all odd-numbered lanes. Conversely, the SUBADD instruction performs subtraction in all even-numbered lanes and addition in all odd-numbered lanes. In these instructions the alternating addition and subtraction operations behave identically to the elemental operations performed by a corresponding F<P>ADD or F<P>SUB instruction.

37.4.2.1 Vector Permutation Operations

Dual-operand vector permutation operations combine two FP vector source operands using one of three permutation functions - performing packing, unpacking and butterfly operations. Single-operand vector permutation operations return a permutation of the elements of a single source vector.

When the `-fp_wide_option` is true, each operand is a pair of FP registers, in common with all vector FP instructions.

Permutation instructions, in common with other FP instructions, are parameterized by the elemental datatype parameter <P>. This defines the element bit width B (half: *B=16*, single: *B=32*, double: *B=64*). The inclusion of vector permutation instructions for each <P> is defined by the following rules:

- By default, all datatypes are supported for all vector permutation operations, unless they are excluded by one of the two rules given below (in which case they raise an Illegal Instruction exception).
- When `-fp_hp_option` is false all permutations where <P> indicates half-precision are excluded.

- If an instruction defines *B* that is equal to `VFP_WIDTH`, then each operand will contain only a single element. In this case the corresponding permutation instruction is redundant, and it is excluded. For example, when VFP_WIDTH = 32 all permutations where <P> selects single-precision are excluded, and likewise, when VFP_WIDTH = 64 all permutations where <P> selects double-precision are excluded.

The number of elements per vector operand is *E* = VFP_WIDTH / B. The number of elements involved in each vector permutation is given by *N*, which depends on the number of operands; for DOP instructions *N* = *2E*, whereas for SOP instructions *N* = *E*. The elements of a vector operand are indexed by integers from 0 to *N*-1, and hence the number of bits needed to represent each element index is given by $n = \log_2(N)$. The largest value of *N* is 16, and occurs for DOP instructions when VFP_WIDTH = 128 and *B* = 16. Hence 1 less than or equal to *n* less than or equal to 4.

Permutation instructions are defined in terms of the mapping from source element indices to destination element indices. We denote the source element index by the *n*-bit value *S*, and denote the destination element index by the *n*-bit value *D*. For each permutation instruction, any source index S maps to a unique destination index *D* according to its permutation function, as listed in Table 37-12.

Dual-operand (DOP) permutations combine two source vectors but are able to deliver only one result vector; hence we provide two versions of each of these DOP instructions. One returns the least-significant half of the result vector, and is suffixed by "L", that is it returns result elements with indices $D < N/2$. The other returns the most-significant half of the result vector, and is suffixed by "M", that is it returns result elements with indices *D* greater than or equal to *N/2*.

Table 37-12 Summary of Vector Permutation Instructions and Their Semantics

Instructions	Permutation	SOP/DOP	Element index mapping function: S -> D
VF<P>PACKL	Shuffle	DOP	D is the left circular rotation of S; i.e. D = {s_{n-2}, …, s_0, s_{n-1}}
VF<P>PACKM			
VF<P>UNPKL	Inverse shuffle	DOP	D is the right circular rotation of S; i.e., D = {s_0, s_{n-1}, …, s_1}
VF<P>UNPKM			
VF<P>BFLYL	Butterfly	DOP	D is a copy of S with bits s_0 and s_{n-1} interchanged; i.e.; D = {s_0, s_{n-2}, …, s_1, s_{n-1}}
VF<P>BFLYM			
VF<P>EXCH	Exchange	SOP	D is a copy of S with bit s0 toggled; i.e.; D = {s_{n-1}, s_{n-2}, …, s_1, s_0 }

The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in Table 37-13. This shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision will permute the source operands in pairs of 16-bit values, which is shown in the table by cell borders encompassing a pair of 16-bit source elements. Likewise, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Shaded cells correspond to instructions that are not supported, and which will raise an Illegal Instruction exception if they are attempted.

Table 37-13 Enumerated Mappings, from Source to Destination, for all Vector Permutation Instructions

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32 (shaded cells represent illegal instructions)															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
vfhexch	b6	b7	b4	b5	b2	b3	b0	b1		b2	b3	b0	b1		b0	b1
vfsexch	b5	b4	b7	b6	b1	b0	b3	b2		b1	b0	b3	b2			
vfdexch	b3	b2	b1	b0	b7	b6	b5	b4								
vfhunpkl	c6	c4	c2	c0	b6	b4	b2	b0		c2	c0	b2	b0		c0	b0
vfhunpkm	c7	c5	c3	c1	b7	b5	b3	b1		c3	c1	b3	b1		c1	b1
vfsunpkl	c5	c4	c1	c0	b5	b4	b1	b0		c1	c0	b1	b0			
vfsunpkm	c7	c6	c3	c2	b7	b6	b3	b2		c3	c2	b3	b2			
vfdunpkl	c3	c2	c1	c0	b3	b2	b1	b0								
vfdunpkm	c7	c6	c5	c4	b7	b6	b5	b4								
vfhpackl	c3	b3	c2	b2	c1	b1	c0	b0		c1	b1	c0	b0		c0	b0
vfhpackm	c7	b7	c6	b6	c5	b5	c4	b4		c3	b3	c2	b2		c1	b1
vfspackl	c3	c2	b3	b2	c1	c0	b1	b0		c1	c0	b1	b0			
vfspackm	c7	c6	b7	b6	c5	c4	b5	b4		c3	c2	b3	b2			
vfdpackl	c3	c2	c1	c0	b3	b2	b1	b0								
vfdpackm	c7	c6	c5	c4	b7	b6	b5	b4								
vfhbflyl	c6	b6	c4	b4	c2	b2	c0	b0		c2	b2	c0	b0		c0	b0
vfhbflym	c7	b7	c5	b5	c3	b3	c1	b1		c3	b3	c1	b1		c1	b1
vfsbflyl	c5	c4	b5	b4	c1	c0	b1	b0		c1	c0	b1	b0			
vfsbflym	c7	c6	b7	b6	c3	c2	b3	b2		c3	c2	b3	b2			
vfdbflyl	c3	c2	c1	c0	b3	b2	b1	b0								
vfdbflyM	c7	c6	c5	c4	b7	b6	b5	b4								

37.4.3 Floating-point Comparison Instructions

The ARCv3 floating-point comparison instructions in Table 37-11 are identical to those implemented in ARCv2. They are encoded in the DOP format, which has a destination register field (fd). However, the

F<P>CMP and F<P>CMPF instructions do not use the fd field, and therefore for these instruction fd is a reserved field.

If one operand of a floating-point MIN or MAX function is a NaN, then the instruction returns the non-NaN operand as the result, as required by the IEEE floating-point standard.

37.4.4 Unconditional Single-operand Floating-point Instructions

The unconditional single-operand floating-point instructions are listed in Table 37-14. The F<P>SQRT operation computes the square-root of its operand, together with its vectorized version VF<P>SQRT. Square-root operations are enabled only if the -fp_div_option is enabled. The vector square-root operations also require -fp_vec_option to be enabled. The vector exchange permutation is listed in Table 37-12 and its input-output mapping is detailed in Table 37-13.

Table 37-14 Unconditional Single-Operand Floating-point Instructions

Instruction	sopf [1:0]	Operation
F<P>SQRT	0x0	fd = sqrt (fs1)
VF<P>SQRT	0x1	vectorized F<P>SQRT
VF<P>EXCH	0x2	vector exchange permutation
-	0x3	(unused and illegal)

37.4.5 Conditional Floating-point Move Operations

The conditional single-operand floating-point instructions are listed in Table 37-15. This format supports the conditional move instruction, in both scalar and vector forms. The F<P>MOV.<cc> instruction examines the selected ARC condition code, and if true the fs1 source register is copied to the destination register fd, otherwise no registers are updated. The vectorized form applies the same conditional test to all elements of the source vector fs1, and either copies all elements or none (depending on the condition code).

Note The F<P>MOV.<cc> and VF<P>MOV.<cc> instructions support only baseline conditions. This restriction means you cannot use an extension condition implemented using the APEX mechanism as the <cc> operand in these instructions.

Table 37-15 Conditional Floating-point Instructions

Instruction	sopf [1:0]	Operation
F<P>MOV.<<cc>>	0x0	fd = (cond(cc) == true) ? fs1
VF<P>MOV.<<cc>>	0x1	vectorized F<P>MOV.<cc>
-	0x2 - 0x3	(reserved for future expansion)

37.4.6 Floating-point Type Conversion Instructions

The `FP_CVF2F` format encodes all floating-point type conversion operations using a 2-bit function field (cvtf) in bits [15:14], and a 5-bit literal (U5) in bits [4:0]. The `cvtf` field encodes the four existing (*that is,* ARCv2) `FCVTx_y` instructions (where x, y can be 32 or 64). The U5 literal encodes the literal operand of those instructions using bits [4:0]; these bits define the source and destination types.

Note The ARCv3 ISA therefore provides a directly equivalent set of floating-point type conversion operations to those previously supported in ARCv2.

The `FVCT32` instruction also supports conversions to/from half-precision values and therefore remains functionally compatible with current and future, scalar and vector, FPU implementations in HS and EV products.

The type-conversion format also defines two register operands; a source and a destination. Note, these are each 5-bit fields, capable of specifying either an integer register in the range `r0-r31` or a floating-point register in the range `f0-f31`. This means that extension core registers cannot be used with floating-point conversion functions in the ARCv3 ISA.

The full set of conversion operations is shown in Table 37-16. This relates the U[4:0] operand to the conversion functions defined by each of the four instructions FCVT32, FCVT32_64, FCVT64_32, and FCVT64.

Table 37-16 Floating-point Conversion Instructions Enumerated

	U5 Operand					FCVT Instruction Variants			
						FCVT32	FCVT32_64	FCVT64_32	FCVT64
ARCv3 Conversion Operation	4	3	2	1	0	cvtf = 00	cvtf = 01	cvtf = 10	cvtf = 11
Unsigned integer to FP type	0	0	0	0	0	FUINT2S	FUINT2D	FUL2S	FUL2D
FP type to unsigned int (use RM to round)	0	0	0	0	1	FS2UINT	FS2UL	FD2UINT	FD2UL
Signed integer to FP type	0	0	0	1	0	FINT2S	FINT2D	FL2S	FL2D
FP type to signed int (use RM to round)	0	0	0	1	1	FS2INT	FS2L	FD2INT	FD2L
FP type 1 to FP type 2 (use RM to round)	0	0	1	0	0	-	FS2D	FD2S	-
-	0	0	1	0	1	-	-	-	-
Round FP type to integral (RM rounding)	0	0	1	1	0	FSRND	-	-	FDRND
-	0	0	1	1	1	-	-	-	-
-	0	1	0	0	0	-	-	-	-
FP type to unsigned int (round to zero)	0	1	0	0	1	FS2UINT_RZ	FS2UL_RZ	FD2UINT_RZ	FD2UL_RZ
-	0	1	0	1	0	-	-	-	-
FP type to signed int (round to zero)	0	1	0	1	1	FS2INT_RZ	FS2L_RZ	FD2INT_RZ	FD2L_RZ
-	0	1	1	0	0	-	-	-	-
-	0	1	1	0	1	-	-	-	-
Round FP type to integral (round to zero)	0	1	1	1	0	FSRND_RZ	-	-	FDRND_RZ
-	0	1	1	1	1	-	-	-	-

Table 37-16 Floating-point Conversion Instructions Enumerated

ARCv3 Conversion Operation	U5 Operand 4	3	2	1	0	FCVT Instruction Variants FCVT32 cvtf = 00	FCVT32_64 cvtf = 01	FCVT64_32 cvtf = 10	FCVT64 cvtf = 11
Move GPR to FPR (no format conversion)	1	0	0	0	0	FMVI2S	-	-	FMVL2D
Move FPR to GPR (no format conversion)	1	0	0	0	1	FMVS2I	-	-	FMVD2L
-	1	0	0	1	0	-	-	-	-
-	1	0	0	1	1	-	-	-	-
Single to half precision (use RM to round)	1	0	1	0	0	FS2H	-	-	-
Half to single precision (no rounding)	1	0	1	0	1	FH2S	-	-	-
-	1	0	1	1	0	-	-	-	-
-	1	0	1	1	1	-	-	-	-
-	1	1	0	0	0	-	-	-	-
-	1	1	0	0	1	-	-	-	-
-	1	1	0	1	0	-	-	-	-
-	1	1	0	1	1	-	-	-	-
-	1	1	1	0	0	-	-	-	-
Single to half precision (round to zero)	1	1	1	0	1	FS2H_RZ	-	-	-
-	1	1	1	1	0	-	-	-	-
-	1	1	1	1	1	-	-	-	-

Table 37-17 Summary of FMV* Instructions and their Semantics

ISA	-fp_dp_option	-fp_vec_option && -fp_wide_option	Instruction	Semantics	Bits moved	GPRs used	FPRs used
ARC64	False	X	FMVI2S	Fd ß Rs [31:0]	32	1	1
			FMVS2I	Rd ß { 0x0000_0000, Fs }	32	1	1
		False	FMVL2D	Raises Illegal Instruction	-	-	-
			FMVD2L	Raises Illegal Instruction	-	-	-
		True	FMVL2D	{ Fd+1, Fd } ß Rs	64	1	2
			FMVD2L	Rd ß { Fs+1, Fs }	64	1	2
	True	X	FMVI2S	Fd ß { 0x0000_0000, Rs [31:0] }	32	1	1
			FMVS2I	Rd ß { 0x0000_0000, Fs [31:0] }	32	1	1
			FMVL2D	Fd ß Rs	64	1	1
			FMVD2L	Rd ß Fs	64	1	1

The configuration of `FCVT` instructions in ARCv3 follows the approach defined in ARCv2. The FCVT32 instruction is included whenever `-has_fp == true`. The remaining three conversion instructions are included only if `-fp_dp_option== true`. Note, the conversions between half-precision and single-precision are included when `-has_fp == true`, regardless of whether `-fp_hp_option` is `true` or `false`.

There are four new instructions to round a floating-point value to an integral value without changing its format. The FSRND instruction rounds a single-precision value to the nearest integer value, using the rounding mode defined by the RM bits in FP_CTRL. Likewise, the FDRND instruction rounds a double-precision value in the same way. There are equivalent instructions, FSRND_RZ and FDRND_RZ, that round towards zero when determining the nearest integer value.

Any attempt to execute a format-conversion instruction that is not supported in a given configuration, or which specifies an unused encoding, raises an Illegal Instruction exception.

37.4.6.1 Moving Values between GPRs and FPRs

There are four instructions for moving values, without type conversion, between integer and floating-point register files (FMVI2S, FMVL2D, FMVS2I and FMVD2L). The cvtf opcode is 2'b00 for FMVI2S and FMVS2I, and 2'b11 for FMVL2D and FMVD2L instructions.

When -fp_dp_option is false in an ARC64 core, a transfer to a 64-bit GPR from a 32-bit FPR (FMVS2I) sets the lower 32 bits of the destination GPR and clears the upper 32 bits, just like other 32-bit instructions in ARC64. Likewise, an ARC64 transfer from a 64-bit GPR to a 32-bit FPR (FMVI2S) copies the lower 32 bits of the GPR to the FPR. If this instruction is performed in a core with DP floating-point, it clears the upper 32 bits of the 64-bit destination FPR.

Any attempt to use an odd-numbered register pair as the integer operand of a floating-point convert of move operation will raise an Illegal Instruction exception.

Distinct FMV* instructions are defined for moves involving values that are single-precision or smaller (FMVI2S, FMVS2I) and values that are double-precision (FMVL2D, FMVD2L). The semantics of all FMV* instructions is summarized in Table 37-17. In Table 37-17 Fd (respectively Rd) represents the destination floating-point register d (respectively general-purpose core register d), and Fs (respectively Rs) represents the source floating-point register s (respectively general-purpose core register s). A wide vector of floating-point registers comprises a register pair, and specified as { Fs+1, Fs } when used as a source operand, or { Fd+1, Fd } when used as a destination operand.

When an ARC64 core is configured with single-precision floating-point with support for wide vector operations, that is, when -fp_dp_option is false but -fp_vec_option and -fp_wide_option are both true, the FMVL2D and FMVD2L instructions move a vector of two single-precision floating-point values given by an even-numbered pair of 32-bit FP registers, to or from a 64-bit core register. However, these two operations are illegal in configurations where -fp_dp_option is false and at least one of the options -fp_vec_option and/or -fp_wide_option are false.

An "X" in the third columns of indicates configurations where the option -fp_vec_option and the option -fp_wide_option are not relevant.

37.4.7 Floating-Point Vector/Scalar Moves

There are two sub-formats of `F32_FP_OPS` that encode instruction to insert or extract scalar elements of a short floating-point vector contained in a short floating-point vector; these are `FP_VMVI` and `FP_VMVR`. The operations encoded in these formats are summarized Table 37-18.

Table 37-18 FP Scalar Insertion, Extraction, and Replication

Instruction	Operands	Sub-format	vmvf	Description
VF<P>EXT	fd, fs1 [u5]	FP_VMVI	0x0	Extract scalar from vector at literal index
	fd, fs1 [b]	FP_VMVR		Extract scalar from vector at variable index
VF<P>INS	fd [u5], fs1	FP_VMVI	0x1	Insert scalar into vector at literal index
	fd [b], fs1	FP_VMVR		Insert scalar into vector at variable index
VF<P>REP	fd, fs1	FP_VMVI (u5 = 0)	0x2	Replicate scalar in all vector element positions
-	-	FP_VMVI (u5 !=0)		(unused and illegal)
-	-	FP_VMVR		(unused and illegal)
-	-	FP_VMVI	0x3	(unused and illegal)
-	-	FP_VMVR		(unused and illegal)

Example 1: `VFHEXT f0, f4[3]` (extract half-precision element 3 from f4 and assign to f0)

Example 2: `VFSINS f2[r0], f12` (insert single-precision f12 value into f2 at index given by r0

Example 3: `VFHREP f4, f0` (replicate half-precision value from f0 across all HP elements of f4)

The number of consecutive vector elements contained in a floating-point register is a function `VLEN`, which depends on `VFP_WIDTH` and the size of the floating-point type given by the P operand. That is: `VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P)`.

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, then each source or destination vector operand for these vector insert/extract/replicate operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

The VF<P>EXT and VF<P>INS instructions silently ignore any bits of the u5 or B-register index operand that are not needed to index into the vector operand. Effectively, the index operand is always taken modulo VLEN before being used as the index.

It is not illegal to perform insert, extract or replicate operations when VLEN(VFP_WIDTH, P) = 1. Such degenerate cases are simple moves from one vector register to another.

37.5 Floating-point MIN and MAX Operation Semantics

The ARCv3 floating-point ISA is designed to conform to the IEEE 754-2008 standard, and hence the F<P>MIN and F<P>MAX instructions conform to the 754-2008 behavior of minNum and maxNum operations. In common with most other implementations of min/max operations under this version of the standard, ARCv3 propagates sNaN values through FP max/min operations as a qNaN, and treats a qNaN as "missing data". The behavior of FP min/max operations is enumerated in the following table:

Operand1	Operand2	Result Value	IE Signaling
sNaN1	sNaN2	Quieted (sNaN1)	Y
sNaN1	qNaN2	Quieted (sNaN1)	Y
sNaN1	Norm2	Quieted (sNaN1)	Y
qNaN1	sNaN2	Quieted (sNaN2)	Y
qNaN1	qNaN2	qNaN1	N
qNaN1	Norm2	Norm2	N
Norm1	sNaN2	Quieted (sNaN2)	Y
Norm1	qNaN2	Norm1	N

The first two columns enumerate all combinations of notable values of each operand when at least one is a NaN. The third column specifies the result that is returned in each case, and the fourth column indicates whether an Invalid Exception is signaled by the operation (by the FPU_STATUS).

A signaling NaN is "quieted" by setting the most-significant bit of the significant field to 1. All other bits of a quieted sNaN operand are passed to the result unmodified.

When both operands of an FP min/max operation are zeros but with different signs, max returns +0.0 and min returns -0.0, that is. -0.0 is considered to be "smaller" than +0.0.

37.6 Accumulator Architecture

When a single 32-bit accumulator is read, the lower 32 bits of the ACC0 register supply the value. When a single 32-bit accumulator is written, then lower 32 bits of ACC0 receive the result, and the upper 32 bits receive either an extension of the sign bit ACC0[31] if ACC0 was written by a signed operation, or else zeros.

ARC64 integer multiply/MAC operations that reference a 64-bit accumulator use ACC0.

38 Floating-Point Auxiliary Registers

Table 38-1 FPU Registers

Address	Auxiliary Register Name
0x300	Floating-Point Unit Control Register, FP_CTRL
0x301	Floating-Point Unit Status Register, FPU_STATUS
0x306	Floating-Point Unit Vector Status Register, VFPU_STATUS
0x310	Floating-Point Unit File Address Register, FP_RF_ADDR
0x311	Floating-Point Register File Data0 Register, FP_RF_DATA0
0x312	Floating-Point Register File Data 1 Register, FP_RF_DATA1
0xC8	Floating-Point Unit Build Register, FPU_BUILD

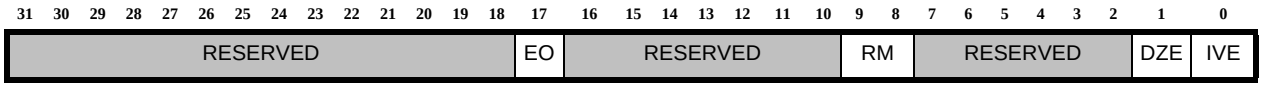
38.1 Floating-Point Unit Control Register, FP_CTRL

Address: 0x300

Access: rw

Default 0x00000100

Figure 38-1 FP_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.

Note When a legacy ARCv2 floating-point unit is configured, the FP_CTRL register follows the layout defined in the *ARCv3 ISA Programmers Reference Manual for ARC HS Cores*.

Table 38-2 FP_CTRL Field Description

Field	Bits	Description
IVE	0	Enable Invalid Operation Exception ■ `0x0:` An exception is not raised when an invalid operation occurs. However, the IV flag in the FPU_STATUS register is set to `1`. ■ `0x1:` An exception is raised when an invalid operation occurs, and in this case the IV flag in the FPU_STATUS register is unchanged.
DZE	1	Enable Divide-by-Zero Exception ■ `0:` An exception is not raised when a divide-by-zero operation occurs. However, the DZ flag in the FPU_STATUS register is set to `1`. ■ `1:` Exception is raised when a divide-by-zero operation occurs and in this case the DZ flag in the FPU_STATUS register is unchanged. This exception is raised only by the hardware divider implementation. A software divider implementation does not raise this exception.

Table 38-2 FP_CTRL Field Description

Field	Bits	Description
RM	[9:8]	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.
EO	[17]	Exception Ordering ■ `0x0`: Allow floating-point exceptions to be taken out-of-order with respect to non floating-point instructions. Note: All floating-point exceptions are taken in the order the originating floating-point instructions were issued. ■ `0x1`: Force all floating-point exceptions to be taken in strict program order. When the OE bit is set, a floating-point exception is taken immediately after the floating-point instruction that triggers the exception but before any subsequent instruction is executed. Note: There may be a significant performance penalty when this feature is enabled. This feature is primarily to enable floating-point exceptions to be reported precisely during software debugging.

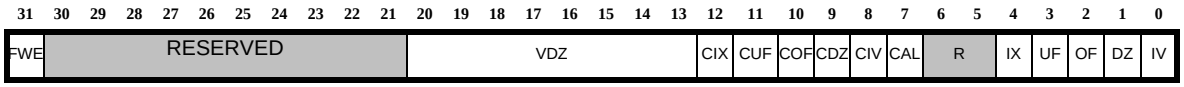
38.2 Floating-Point Unit Status Register, FPU_STATUS

Address: 0x301

Access: rw

Reset 0x00000000

Figure 38-2 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 when each flag-modifying floating-point operation. On processor reset, all the flags are cleared.

Processors configured with vector floating-point divide and square-root capability, that is, `-fp_vec_option` and `-fp_div_option` are enabled, implement the `VDZ` field. The `VDZ` field does not exist and is reserved in the processor configurations that do not support the floating-point divide operations or do not support the vector floating-point capability. In this case, the `VDZ` field (bits [20:13]) are reserved.

Scalar floating-point instructions update only the scalar status flags `IX`, `UF`, `OF`, `DZ`, and `IV`. When configured, vector floating-point instructions update both vector and scalar status flags. Vector operations set each sticky scalar flag with the logical OR of the corresponding status result from each vector operation element. For example, if any basic operation overflows, then the `OF` bit is set.

When flags are cleared by writing `1` to bits in the range [12:7] of the `FPU_STATUS` register, the corresponding scalar and vector status flags are cleared. The `VFPU_STATUS` register that includes the vector status flags exists in the processor when the vector floating-point capabilities are configured. For example, write `1` to the FPU_STATUS.CIV bit to clear FPU_STATUS.IV and all bits in VFPU_STATUS.VIV.

Table 38-3 FPU_STATUS Field Description

Field	Bits	Description
IV	0	Invalid operation sticky flag. This flag is updated implicitly based on the result of a scalar floating-point operation. This flag is a sticky flag. You can clear the flag in the following ways: ■ 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. Write `1` to this clear this flag.
DZ	1	Divide-by-zero sticky flag. This flag is updated implicitly based on the result of a scalar floating-point operation. This flag is a sticky flag. You can clear the flag in the following ways: ■ 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. Write `1` to this clear this flag.
OF	2	Overflow sticky flag. This flag is updated implicitly based on the result of a scalar floating-point operation. This flag is a sticky flag. You can clear the flag in the following ways: ■ 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. Write `1` to this clear this flag.

Table 38-3 FPU_STATUS Field Description

Field	Bits	Description
UF	3	Underflow sticky flag. This flag is updated implicitly based on the result of a scalar floating-point operation. This flag is a sticky flag. You can clear the flag in the following ways: ■ 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. Write 1 to this clear this flag.
IX	4	Inexact sticky flag. This flag is updated implicitly based on the result of a scalar floating-point operation. This flag is a sticky flag. You can clear the flag in the following ways: ■ 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. Write 1 to this clear this flag.
CAL	7	Clear all flags. ■ If FWE==0, you can write to this bit. Writes to this bit clear bits [4:0] of FPU_STATUS and all bits of VFPU_STATUS when -fp_vec_option is enabled. ■ If FWE==1, this bit read as zero and is ignored on write.
CIV	8	Clear invalid operation flag. ■ If FWE==0, you can write to this bit. Writes to this bit clear the scalar bit and all bits of the VFPU_STATUS.VIV field when -fp_vec_option is enabled. ■ If FWE==1, this bit read as zero and is ignored on write.
CDZ	9	Clear divide by zero. ■ If FWE==0, you can write to this bit. Writes to this bit clear the FPU_STATUS.DZ bit and all bits of the scalar field when scalar is enabled. ■ If FWE==1, this bit read as zero and is ignored on write.
COF	10	Clear overflow flag. ■ If FWE==0, you can write to this bit. Writes to this bit clear the FPU_STATUS.OF bit and all bits of the FPU_STATUS.VOF field when -fp_vec_option is enabled. ■ If FWE==1, this bit read as zero and is ignored on write.
CUF	11	Clear underflow flag. ■ If FWE==0, you can write to this bit. Writes to this bit clear the FPU_STATUS.UF bit and all bits of the FPU_STATUS.VUF field when -fp_vec_option is enabled. ■ If FWE==1, this bit read as zero and ignored on write.
CIX	12	Clear inexact flag. ■ If FWE==0, you can write to this bit. Writes to this bit clear the FPU_STATUS.IX bit and all bits of the FPU_STATUS.VIX field when -fp_vec_option is enabled. ■ If FWE==1, this bit read as zero and ignored on write.
VDZ	[20:13]	Divide-by-zero sticky flags vector. This field is implemented only if the vector floating-point feature is enabled by -fp_vec_option. This field is then updated implicitly when a vector FP divide operation is executed.

Table 38-3 FPU_STATUS Field Description

Field	Bits	Description
FWE	31	Flag Write Mode; The FWE bit is not stored in the FPU_STATUS register and its value is only relevant when writing to FPU_STATUS, when it acts as a control signal to determine whether individual flags are cleared or all stored fields of register are written. The interpretation of this bit is: ■ 0: Disables direct writes to status fields in the FPU_STATUS register. In this mode, status flags can be cleared only by writing 1 to CAL or to a selected subset of the CIV, CDZ, COF, CUF, and CIX fields. FPU_STATUS bits [13:7] are write-only in this mode and bits [4:0] and [23:16] are read-only. ■ 1: Enables direct writes to status fields in the FPU_STATUS register. In this mode, status write data is copied to the IV, DZ, OF, UF, and IX fields of FPU_STATUS. FPU_STATUS bits [13:7] read as zero in this mode and bits [4:0] and [23:16] are write-only.

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 FP_CTRL.IVE==0 (invalid operation exception 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.

UF — Underflow

Underflow occurs when the result is too small to be represented by the instruction data format.

IX —Inexact

Inexact operation occurs when the result of a floating-point operation is rounded and not exact.

The invalid operation flag is set according to the definition of each instruction in Chapter 39, "Floating-Point Unit Instructions".

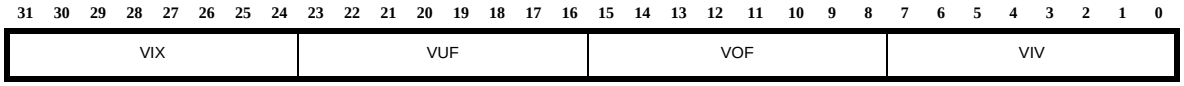
38.3 Floating-Point Unit Vector Status Register, VFPU_STATUS

Address: 0x306

Access: RW

Reset 0x00000000

Figure 38-3 VFPU_STATUS Register



This register is present only when `-fp_vec_option` is enabled.

This register allows you to read the status of the vector floating-point unit flags. These flags are updated on the completion of each flag-modifying vector floating-point operation. On processor reset, all flags are cleared. Each group of vector floating-point flags is cleared when its corresponding scalar flag is cleared in the FPU_STATUS register.

A flag-setting floating-point instruction that operates on a vector of *N* elements updates only elements 0 to *N-1* of each `VFPU_STATUS` field. For example, a VFSADD instruction on a processor with double-precision support but without wide vectors, operates on vectors of two elements, and thus update bits [1:0] of each `VFPU_STATUS` field. However, in a maximal vector floating-point configuration, a VFHADD instruction operates on vectors of eight elements and thus update bits [7:0] of each `VFPU_STATUS` field.

A processor configuration that supports vector floating-point operations can support vector operations over a maximum width `VFP_MAX_VLEN`. If `VFP_MAX_VLEN` is less than the maximum value 8, then bits [7: VFP_MAX_VLEN-1] of the `VIX`, `VUG`, `VOF`, `VIV`, and `FPU_STATUS.VDZ` fields are reserved. These bits return `0` when read and writes are ignored.

Table 38-4 lists the field descriptions:

Table 38-4 VFPU_STATUS Field Description

Field	Bits	Description
VIV	[7:0]	**Invalid operation sticky flag vector** This field is implemented only if the vector floating-point feature is enabled by `-fp_vec_option`. The elements of this vector are then set implicitly if the corresponding elemental operations of a vector FP instruction are invalid.
VOF	[15:8]	**Overflow sticky flag vector** This field is implemented only if the vector floating-point feature is enabled by `-fp_vec_option`. The elements of this vector are then set implicitly if the corresponding elemental results of a vector FP instruction overflow.

Table 38-4 VFPU_STATUS Field Description (Continued)

Field	Bits	Description
VUF	[23:16]	**Underflow sticky flag vector** This field is implemented only if the vector floating-point feature is enabled by `-fp_vec_option`. The elements of this vector are then set implicitly if the corresponding elemental results of a vector FP instruction underflow.
VIX	[31:24]	**Inexact sticky flag vector** This field is implemented only if the vector floating-point feature is enabled by `-fp_vec_option`. The elements of this vector are then set implicitly if the corresponding elemental results of a vector FP instruction are inexact.

38.4 Floating-Point Unit File Address Register, FP_RF_ADDR

Address: 0x310

Access: RG

Default 0x00000000

Figure 38-4 FP_RF_ADDR Register

Use this register to access a floating-point register. When this register is written with an address value, the data value of the floating point register located at that address is loaded into FP_RF_DATA0, FP_RF_DATA1.

Table 38-5 FP_RF_ADDR Description

Field	Bits	Description
ADDR	[4:0]	ADDR field in this register holds the address value of floating point register to be read or write. The width of the ADDR field depends on the number of floating point registers configured with `-fp_num_regs` option. For the number of registers configured as 8, 16, 32, width of the ADDR field width will be 3, 4, or 5 bits respectively.
RESERVED	[31:5]	Reserved

Reading FP register *N*

1. Write *N* to the FP_RF_ADDR register.
2. Read data bits [63:32] from FP_RF_DATA1 if a double-precision (DP) register is being read, otherwise skip this step.
3. Read data bits [31:0] from FP_RF_DATA0.

Writing FP register *N*

1. Write *N* to the FP_RF_ADDR register.
2. Write data bits [63:32] to FP_RF_DATA1 if double-precision (DP) floating-point is configured, otherwise skip this step. If this step is skipped when DP is configured, the write uses the previous value in the FP_RF_DATA1 register; this can be used to optimize DP writes where the upper 32 bits of several registers are written with the same value.
3. Write data bits [31:0] to FP_RF_DATA0. This operation triggers the writing of all data to the designated FP register. If DP is configured, a 64-bit write operation is triggered: FP_RF_DATA1 to

bits [63:32] of the FP register selected by FP_RF_ADDR, and FP_RF_DATA0 to bits [31:0] of that register. If DP is not configured, a 32-bit write operation is triggered: FP_RF_DATA0 to bits [31:0] of the FP register selected by FP_RF_ADDR.

Note

The core ensures that any pending floating-point operation has completed before any auxiliary access to these registers is performed, potentially stalling LR, SR, or AEX instructions for a few cycles until this condition is met.

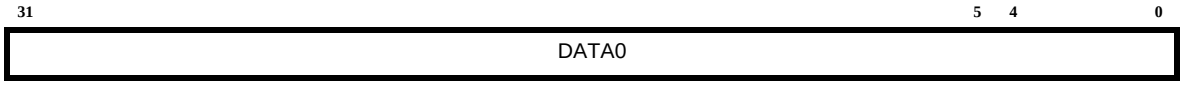
38.5 Floating-Point Register File Data0 Register, FP_RF_DATA0

Address: 0x311

Access: RG

Default 0x00000000

Figure 38-5 FP_RF_DATA0 Register



This register is used as an interface to write a data value or read a data value from a floating point register.

When data is written to this register, a write is initiated to the floating-point register at the address specified by the FP_RF_ADDR register.

When data is read from this register, a read is initiated to the floating-point register at the address specified by the FP_RF_ADDR register.

38.6 Floating-Point Register File Data 1 Register, FP_RF_DATA1

Address: 0x312

Access: RG

Default 0x00000000

Figure 38-6 FP_RF_DATA1 Register

This register is used as an interface to write a data value or read a data value from a floating point register.

This register is present if double-precision (DP) floating-point is configured. The upper 32 bits, that is, [63:32] range of floating point register data is accessed with FP_RF_DATA1.

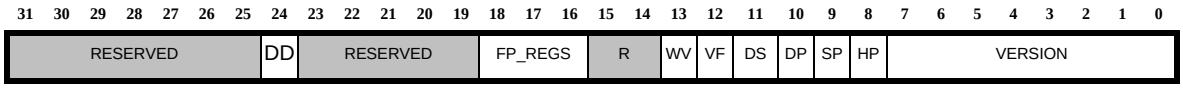
For performing a 64-bit write into floating point register, FP_RF_DATA1 register has to be written before writing FP_RF_DATA0.

38.7 Floating-Point Unit Build Register, FPU_BUILD

Address: 0xC8

Access: R

Figure 38-7 FPU_BUILD Register



Note When a legacy ARCv3 floating-point unit is configured, the FPU_BUILD register follows the layout defined in the *ARCv3 ISA Programmers Reference Manual for ARC HS Cores*.

This register is present only if a floating-point unit is included in the processor configuration. Table 38-6 lists the field descriptions:

Table 38-6 FPU_BUILD Field Description

Field	Bits	Description
VERSION	[7:0]	Floating-Point Unit Version `0x04`:ARCv3 floating-point unit is configured
HP	8	Indicates support for half-precision floating-point operations, that is when the `-fp_hp_option` is true.
SP	9	Indicates support for single-precision floating-point operations, that is when `-has_fp` is true.
DP	10	Indicates support for double-precision floating-point operations, that is when the `-fp_dp_option` is true.
DS	11	Indicates support for divide and square-root operations, that is when the `-fp_div_option` is true.
VF	12	Indicates support for vector floating-point operations, that is when the `-fp_vector_option` is true.
WV	13	When WV is 1, this bit indicates support for wide vector operations. that is when the `-fp_wide_option` is true.
FP_REGS	[18:16]	Log_2 of the number of configured floating-point registers. It is always one of the values either 3, 4 or 5.
DD	24	Indicates support for demand-driven floating-point state save and restore functionality, that is when the `-fp_dds_option` is true.

39 Floating-Point Unit Instructions

Note When a build configuration includes a floating-point unit, "HAS_FP" is TRUE, otherwise it is FALSE.

Table 39-1 Floating-Point Instructions

Instruction	Operation
FCVT32	Convert between single-precision float, half-precision float, signed, or unsigned integer
FCVT32_64	Convert from 32-bit data formats (single-precision float, signed, or unsigned integer) to 64-bit data formats
FCVT64	Convert between two 64-bit data formats: double-precision float, signed, or unsigned long
FCVT64_32	Convert from 64-bit data formats to 32-bit data formats
FDADD	Double-precision floating-point addition
FDCMP	Double-precision floating-point comparison
FDCMPF	Double-precision floating-point comparison operation with IEEE 754 flag generation
FDDIV	Double-precision floating-point division
FDMADD	Double-precision floating-point fusedMultiplyAdd
FDMAX	Return maximum value of two double-precision floating-point numbers
FDMIN	Return minimum value of two double-precision floating-point numbers
FDMOV	Double-precision floating-point move operation
FDMOVCC	Move contents from the double-precision source register to the destination register if the conditions are met

Table 39-1 Floating-Point Instructions

Instruction	Operation
FDMSUB	Double-precision floating-point fusedMultiplySubtract
FDMUL	Double-precision floating-point multiply
FDNMADD	Double-precision floating-point fusedMultiplyAdd. The negation of the result is stored in the destination register
FDNMSUB	Double-precision floating-point fusedMultiplySubtract. The negation of the result is stored in the destination register
FDSGNJ	Double-precision floating-point move operation
FDSGNJN	Double-precision floating-point negate operation
FDSGNJX	Double-precision floating-point absolute operation
FDSQRT	Double-precision floating-point square-root
FDSUB	Double-precision floating-point subtraction
FHADD	Half-precision floating-point addition
FHCMP	Half-precision floating-point comparison
FHCMPF	Half-precision floating-point comparison operation with IEEE 754 flag generation
FHDIV	Half-precision floating-point division
FHMADD	Half-precision floating-point fusedMultiplyAdd
FHMAX	Return maximum value of two half-precision floating-point numbers
FHMIN	Return the minimum value of two half-precision floating-point numbers
FHMOVCC	Move contents from the half-precision source register to the destination register if the conditions are met
FHMSUB	Half-precision floating-point fusedMultiplySubtract
FHMUL	Half-precision floating-point multiply
FHNMADD	Half-precision floating-point fusedMultiplyAdd. The negation of the result is stored in the destination register
FHNMSUB	Half-precision floating-point fusedMultiplySubtract. The negation of the result is stored in the destination register
FHSGNJ	Single-precision floating-point move operation
FHSGNJN	Single-precision floating-point negate operation
FHSGNJX	Single-precision floating-point absolute operation
FHSQRT	Half-precision floating-point square-root

Table 39-1 Floating-Point Instructions

Instruction	Operation
FHSUB	Half-precision floating-point subtraction
FLDD32	Loads two consecutive values from even-numbered register pair from the specified memory address
FLDD64	Loads two consecutive 64-bit values from the specified memory address
FLD16	Loads a 16-bit value from the specified memory address and places it into the lower 16 bits of the destination floating-point register
FLD32	Loads a 32-bit value from the specified memory address and places it into the lower 32 bits of the destination floating-point register
FLD64	Loads a 64-bit value from the specified memory address
FMVD2L	Move contents of a floating-point register to a general-purpose register without type conversion
FMVI2S	Move contents of a general-purpose register to a single-precision floating-point register without type conversion
FMVL2D	Move contents of a general-purpose register to a floating-point register without type conversion
FMVS2I	Move contents of a floating-point register to a general-purpose register without type conversion
FSADD	Single-precision floating-point addition
FSCMP	Single-precision floating-point comparison
FSCMPF	Single-precision floating-point comparison operation with IEEE 754 flag generation
FSDIV	Single-precision floating-point division
FSMADD	Single-precision floating-point fusedMultiplyAdd
FSMAX	Return maximum value of two single-precision floating-point numbers
FSMIN	Return minimum value of two single-precision floating-point numbers
FSMOVCC	Move contents from the single-precision source register to the destination register if the conditions are met
FSMSUB	Single-precision floating-point fusedMultiplySubtract
FSMUL	Single-precision floating-point multiply
FSNMADD	Single-precision floating-point fusedMultiplyAdd. The negation of the result is stored in the destination register

Table 39-1 Floating-Point Instructions

Instruction	Operation
FSNMSUB	Single-precision floating-point fusedMultiplySubtract. The negation of the result is stored in the destination register
FSSGNJ	Single-precision floating-point move operation
FSSGNJN	Single-precision floating-point negate operation
FSSGNJX	Single-precision floating-point absolute operation
FSSQRT	Single-precision floating-point square-root
FSSUB	Single-precision floating-point subtraction
FSTD64	Stores two 64-bit values from an even-numbered pair of double-precision floating-point source registers, to memory at the specified address
FSTD32	Stores value from an even-numbered register pair to the specified memory location
FST16	Stores a 16-bit value from the lower 16 bits of the designated floating-point source register to memory, at the specified address
FST32	Stores a 32-bit value from the lower 32 bits of the designated floating-point source register to memory, at the specified address
FST64	Stores a 64-bit value from one double-precision floating-point source register, to memory at the specified address
VFDADD	Double-precision floating-point vector addition
VFDADDS	Add a scalar to a vector with double-precision floating-point elements
VFDADDSUB	Double-precision vector floating-point instruction that performs additions in all even-numbered lanes and subtraction in all odd-numbered lanes
VFDDIV	Double-precision vector floating-point division
VFDDIVS	Divide each double-precision floating-point vector element by a scalar
VFHEXCH	Half-precision vector floating-point instruction that performs vector exchange permutation operation
VFDEXCH	Double-precision vector floating-point instruction that performs vector exchange permutation operation
VFSEXCH	Single-precision vector floating-point instruction that performs vector exchange permutation operation
VFDEXT	Extract double-precision element from vector at a literal index or variable index
VFDINS	Insert double-precision element into vector at a literal index or variable index

Table 39-1 Floating-Point Instructions

Instruction	Operation
VFDMADD	Double-precision floating-point vector fusedMultiplyAdd
VFDMADDS	Double-precision floating-point vector and scalar fusedMultiplyAdd
VFDMOVCC	Move contents from all the double-precision elements in the source vector to the destination vector if the conditions are met
VFDMSUB	Double-precision floating-point vector fusedMultiplySubtract
VFDMSUBS	Double-precision floating-point vector and a scalar fusedMultiplySubtract
VFDMUL	Double-precision vector floating-point multiplication
VFDMULS	Multiply each double-precision element vector floating-point number with a scalar
VFDNMADD	Double-precision floating-point vector fusedMultiplyAdd
VFDNMADDS	Double-precision floating-point vector and scalar fusedMultiplyAdd. The negation of the sum is stored in the destination register
VFDNMSUB	Double-precision floating-point vector fusedMultiplySubtract. The negated result is stored in the destination register
VFDNMSUBS	Double-precision floating-point vector and a scalar fusedMultiplySubtract. The negated result is stored in the destination register
VFDREP	Replicate a double-precision value across all double-precision elements of a vector
VFDSQRT	Double-precision floating-point square-root of all the elements in a vector
VFDSUB	Vector double-precision floating-point subtraction
VFDSUBS	Subtract a scalar from a vector with double-precision floating-point elements
VFDSUBADD	Double-precision vector floating-point instruction that performs subtraction in all even-numbered lanes and additions in all odd-numbered lanes
VFHADD	Half-precision floating-point vector addition
VFHADDS	Add a scalar to a vector with half-precision floating-point elements
VFHADDSUB	Half-precision vector floating-point instruction that performs additions in all even-numbered lanes and subtraction in all odd-numbered lanes
VFHDIV	Half-precision vector floating-point division
VFHDIVS	Divide each half-precision floating-point vector element by a scalar
VFHEXT	Extract half-precision element from vector at a literal index or variable index

Table 39-1 Floating-Point Instructions

Instruction	Operation
VFHINS	Insert half-precision element into vector at a literal index or variable index
VFHMADD	Half-precision floating-point vector fusedMultiplyAdd.
VFHMADDS	Half-precision floating-point vector and scalar fusedMultiplyAdd
VFHMOVCC	Move contents from all the half-precision elements in the source vector to the destination vector if the conditions are met
VFHMSUB	Half-precision floating-point vector fusedMultiplySubtract
VFHMSUBS	Half-precision floating-point vector and a scalar fusedMultiplySubtract
VFHMUL	Half-precision vector floating-point multiplication
VFHMULS	Multiply each half-precision element vector floating-point number with a scalar
VFHNMADD	Multiply each half-precision element vector floating-point number with a scalar. The negation of the result is stored in the destination register
VFHNMADDS	Half-precision floating-point vector and scalar fusedMultiplyAdd. The negation of the result is stored in the destination register
VFHNMSUB	Half-precision floating-point vector fusedMultiplySubtract
VFHNMSUBS	Half-precision floating-point vector and a scalar fusedMultiplySubtract. he negation of the result is stored in the destination register
VFHREP	Replicate a half-precision value across all half-precision elements of a vector
VFHSQRT	Half-precision floating-point square-root of all the elements in a vector
VFHSUB	Vector half-precision floating-point subtraction
VFHSUBS	Subtract a scalar from a vector with half-precision floating-point elements
VFHSUBADD	Half-precision vector floating-point instruction that performs subtraction in all even-numbered lanes and additions in all odd-numbered lanes
VFSADD	Single-precision floating-point vector addition
VFSADDS	Add a single-precision floating-point vector with a scalar
VFSADDSUB	Single-precision vector floating-point instruction that performs additions in all even-numbered lanes and subtraction in all odd-numbered lanes
VFSDIV	Single-precision vector floating-point division
VFSDIVS	Divide each single-precision floating-point vector element by a scalar
VFSEXT	Extract single-precision element from vector at a literal index or variable index

Table 39-1 Floating-Point Instructions

Instruction	Operation
VFSINS	Insert single-precision element into vector at a literal index or variable index
VFSMADD	Single-precision floating-point vector fusedMultiplyAdd.
VFSMADDS	A vector is multiplied by a scalar, and the product is added to another vector.
VFSMOVCC	Move contents from all the single-precision elements in the source vector to the destination vector if the conditions are met
VFSMSUB	Single-precision floating-point vector fusedMultiplySubtract
VFSMSUBS	Single-precision floating-point vector and a scalar fusedMultiplySubtract
VFSMUL	Single-precision vector floating-point multiplication
VFSMULS	Multiply each single-precision element vector floating-point number with a scalar
VFSNMADD	Single-precision floating-point vector fusedMultiplyAdd
VFSNMADDS	Single-precision floating-point vector and scalar fusedMultiplyAdd
VFSNMSUB	Single-precision floating-point vector fusedMultiplySubtract
VFSNMSUBS	Single-precision floating-point vector and a scalar fusedMultiplySubtract
VFSREP	Replicate a single-precision value across all single-precision elements of a vector
VFSSQRT	Single-precision floating-point square-root of all the elements in a vector
VFSSUB	Vector single-precision floating-point subtraction
VFSSUBS	Subtract a scalar from a vector with single-precision floating-point elements
VFSSUBADD	Single-precision vector floating-point instruction that performs subtraction in all even-numbered lanes and additions in all odd-numbered lanes
FSABS	Single-precision floating-point absolute operation
FHABS	Half-precision floating-point absolute operation
FDABS	Double-precision floating-point absolute operation
FSNEG	Single-precision floating-point negate operation
FHNEG	Half-precision floating-point negate operation
FDNEG	Double-precision floating-point negate operation

FCVT32

Function

Convert between single-precision float, half-precision float, signed, or unsigned integer.

Alias

FS2INT, FS2UINT, FS2INT_RZ, FS2UINT_RZ, FINT2S, FUINT2S, FS2H_RZ, FH2S, FS2H, FMVS2I, FMVI2S, FSRND_RZ, FSRND

Extension Group

HAS_FP == TRUE

Operation

```
e = (int) m;
e = (float) m;
e = (uint)m;
```

Instruction Format

op e, m, u5

Syntax Example

FCVT32 l,m,u5

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 an integer. or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).

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 from one format to another. The type of conversion is determined by the u5 operand. Depending on the value of u5, the processor does the following conversions:

Instruction	u5 value	Description
FUINT2S	FCVT32 e, m, 00000	Unsigned integer to single-precision float
FS2UINT	FCVT32 e, m, 00001	Single-precision float to unsigned integer
FINT2S	FCVT32 e, m, 00010	Signed integer to single-precision float
FS2INT	FCVT32 e, m, 00011	Single-precision float to signed integer
FSRND	FCVT32 e, m, 00110	Single-precision value to the nearest integer value, using the rounding mode defined by the RM bits in FP_CTRL
FS2UINT_RZ	FCVT32 e, m, 01001	Single-precision float to unsigned integer; round to zero
FS2INT_RZ	FCVT32 e, m, 01011	Single-precision float to signed integer; round to zero
FSRND_RZ	FCVT32 l, m, 01110	Single-precision value to the nearest integer value; round to zero
FMVI2S	FCVT32 e, m, 10000	Move GPR to FPR (no format conversion)
FMVS2I	FCVT32 e, m, 10001	Move FPR to GPR (no format conversion)
FS2H	FCVT32 e, m, 10100	Single precision to half precision; using the rounding mode defined by the RM bits in FP_CTRL to round
FH2S	FCVT32 e, m, 10101	Half precision to single precision with no rounding
FS2H_RZ	FCVT32 e, m, 11100	Single precision to half precision; round to zero

Pseudo Code

```
e = (float) m;                          /* FCVT32 */
```

Assembly Code Example

```
FCVT32 f0,f1, u5   ; conversion from f0 to f1
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FCVT32<.cc>	e,m,u5	11100mmm011000**0000**MM0EEEEE1uuuuu

FCVT32_64

Function

Convert from 32-bit data formats (single-precision float, signed, or unsigned integer) to 64-bit data formats.

Alias

FUINT2D, FS2UL, FINT2D, FS2L, FS2D, FS2UL_RZ, FS2L_RZ

Extension Group

(HAS_FP == TRUE) && (FP_DP_OPTION == TRUE)

Operation

```
E = (long) M;
E = (ulong) M;
E = (double) m;
```

Instruction Format

op e, m, u5

Syntax Example

FCVT32_64 e, m, u5

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).

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 u5 operand. Depending on the value of u5, 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
FUINT2D	FCVT32_64 e, m, 00000	32-bit unsigned integer to double-precision float
FS2UL	FCVT32_64 e, m, 00001	Single-precision float to 64-bit unsigned integer
FINT2D	FCVT32_64 e, m, 00010	32-bit integer to double-precision float
FS2L	FCVT32_64 e, m, 00011	Single-precision float to 64-bit integer
FS2D	FCVT32_64 e, m, 00100	Single-precision float to double-precision float
FS2UL_RZ	FCVT32_64 e, m, 01001	Single-precision float to 64-it unsigned integer; round to zero
FS2L_RZ	FCVT32_64 e, m, 01011	Single-precision float to 64-bit integer; round to zero

Pseudo Code

```
e = (long) (m);                                  /* FCVT32_64 */
```

Assembly Code Example

```
FCVT32_64 f0,f2,u5    ; conversion from f2 to (f1,f0)
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FCVT32_64<.cc>	e,m,u5	11100mmm01100001 00MM0EEEEE1uuuuu

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_FP == TRUE) && (FP_DP_OPTION == TRUE)

Operation

```
e = long m;
e = ulong m;
e = double m;
```

Instruction Format

op e, m, u5

Syntax Example

FCVT64 e,m,u5

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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
FUL2D	FCVT64 e, m, 000000	64-bit unsigned integer to double-precision float
FD2UL	FCVT64 e, m, 000001	Double-precision float to 64-bit unsigned integer
FL2D	FCVT64 e, m, 000010	64-bit integer to double-precision float
FD2L	FCVT64 e, m, 000011	Double-precision float to 64-bit signed integer
FD2UL_RZ	FCVT64 e, m, 001001	Double-precision float to 64-bit unsigned integer; round to zero
FD2L_RZ	FCVT64 e, m, 001011	Double-precision float to 64-bit signed integer; round to zero
FDRND	FCVT64 e, m, 00110	
FDRND_RZ	FCVT64 e, m, 01001	
FMVL2D	FCVT64 e, m, 01011	
FMVD2L	FCVT64 e, m, 10001	

Pseudo Code

```
e = long m;                                  /* FCVT64 */
```

Assembly Code Example

```
FCVT64 f2,f4,u5    ; conversion from (f2,f4)
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FCVT32_64<.cc>	e,m,u5	11100mmm011000**11**00MM0EEEEE1uuuuu

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_FP == TRUE) && (FP_DP_OPTION == TRUE)

Operation

```
e = int m;
e = uint m;
e = float m;
```

Instruction Format

op e, m, u5

Syntax Example

FCVT64_32 e, m, u5

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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
FUL2S	FCVT64_32 e, m, 00000	64-bit unsigned integer to double-precision float
FD2UINT	FCVT64_32 e, m, 00001	Double-precision float to 32-bit unsigned integer
FL2S	FCVT64_32 e, m, 00010	64-bit integer to single-precision float
FD2INT	FCVT64_32 e, m, 00011	Double-precision float to 32-bit integer
FD2S	FCVT64_32 e, m, 00100	Double-precision float to single-precision float
FD2UINT_RZ	FCVT64_32 e, m, 01001	Double-precision float to 32-bit unsigned integer; round to zero
FD2INT_RZ	FCVT64_32 e, m, 01011	Double-precision float to 32-bit integer; round to zero

Pseudo Code

```
e = int m;                                          /* FCVT64_32 */
```

Assembly Code Example

```
FCVT64_32 f0,f2,u5    ; conversion from f2 to (f1,f0)
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FCVT64_32<.cc>	e,m,u5	11100mmm011000**1**000MM0EEEEE1uuuuu

FDADD

Function

Double-precision floating-point addition

Extension Group

FP_DP_OPTION == TRUE

Operation

```
fd = s1 +s2
```

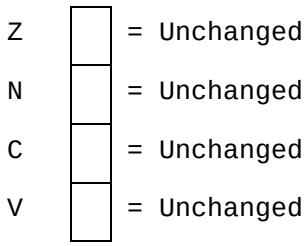
Instruction Format

op fd, s1, s2

Syntax Example

FDADD fd, s1, s2

STATUS32 Flags Affected



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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 sum of s1 and s2 operands is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* FDADD */
```

Assembly Code Example

```
FDADD fd, s1, s2     ; floating-point addition of s1 and s2 and the
                     ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDADD	fd, s1, s2	11100mmm00000000 10MM0EEEEE1LLLLL

FDCMP

Function

Double-precision floating-point comparison

Extension Group

FP_DP_OPTION == TRUE

Operation

```
ZNCV <- FDCMP(s1, s2)
```

Instruction Format

op s1, s2

Syntax Example

FDCMP s1,s2

STATUS32 Flags Affected

Z	•	= Set when s1 and s2 are equal
N	•	= Set when s1 is less than s2
C	•	= Set when s1 is less than s2
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 FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The s1 and s2 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

```
ZNCV <- FDCMP(s1, s2)                                                    /* FDCMP */
```

Assembly Code Example

```
FDCMP s1,s2       ; floating-point comparison of s1 and s2 and the
                  ; core flags are updated
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDCMP	s1,s2	11100mmm00000100 10MM0EEEEE1LLLLL

FDCMPF

Function

Double-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 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

FP_DP_OPTION == TRUE

Operation

```
ZNCV <- FDCMP(s1, s2)    "signaling comparison"
```

Instruction Format

op s1, s2

Syntax Example

FDCMPF s1, s2

STATUS32 Flags Affected

Z	•	= Set when s1 and s2 are equal
N	•	= Set when s1 is less than s2
C	•	= Set when s1 is less than s2
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 FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The s1 and s2 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

```
ZNCV <- FDCMP(s1, s2)    “signaling comparison”            /* FDCMPF */
```

Assembly Code Example

```
FDCMPF s1,s2     ; floating-point comparison of s1 and s2 and update flags
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDCMPF	s1, s2	11100mmm000001011 0MM0EEEEE1LLLLL

FDDIV

Function

Double-precision floating-point division

Extension Group

FP_DP_OPTION == TRUE

Operation

```
fd = (double) s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

FDDIV fd, s1, s2

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 signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ	•	Set when the divisor is zero
OF	•	Set when the normalized result exceeds the maximumrepresentable 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 s1 operand is divided by the s2 operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                                /* FDDIV */
```

Assembly Code Example

```
FDDIV fd,s1,s2     ; floating-point division of s1 by s2 and the
                   ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDDIV	fd, s1, s2	11100mmm00000011 10MM0EEEEE1LLLLL

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

FP_DP_OPTION == TRUE

Operation

```
fd = s3 + (s1 * s2)
```

Instruction Format

op fd, s1, s2, s3

Syntax Example

FDMADD fd, s1, s2, s3

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s3)= +infinity and s1 = -infinity or (s2 * s3)= -infinity and s1 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s2 and s3 operands is computed, added to s1, 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

```
fd = s1 + (s2 * s3)                          /* FDMADD */
```

Assembly Code Example

```
FDMADD fd, s1, s2, s3      ; floating-point multiplication of s2 and s3
                           and the ; product is accumulated and the sum
                           stored in s1.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDMADD	s1, s2, s3	11100mmm00000000 10MM 1EEEEE 0LLLLL

FDMAX

Function

Return maximum value of two double-precision floating-point numbers

Extension Group

FP_DP_OPTION == TRUE

Operation

fd = (s1 > s2) ? s1 : s2

Instruction Format

op fd, s1, s2

Syntax Example

FDMAX fd, s1, s2

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 floating-point operands (s1 and s2) and place the result in the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (s1 > s2) ? s2 : s1                /* FDMAX */
```

Assembly Code Example

```
FDMAX fd, s1, s2    ; Take maximum of s1 and s2 and write result into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDMAX	fd, s1, s2	11100mmm00000111 10MM0EEEEE1LLLLL

FDMIN

Function

Return minimum value of two double-precision floating-point numbers

Extension Group

HAS_FP == TRUE

Operation

fd = (s1 > s2) ? s2 : s1

Instruction Format

op fd, s1, s2

Syntax Example

FDMIN<.f> fd, s1, s2

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 floating-point operands (s1 and s2) and place the result in the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (s1 > s2) ? s2 : s1                /* FDMIN */
```

Assembly Code Example

```
FDMIN fd, s1, s2    ; Take minimum of s1 and s2 and write result into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDMIN	fd, s1, s2	11100mmm00000110 10MM0EEEEE1LLLLL

FDMOV

Function

Double-precision floating-point move operation.

Alias

FDSGNJ f1, f2, f2

FDMOVCC

Function

Move contents from the double-precision source register to the destination register if the conditions are met

Extension Group

FP_DP_OPTION == TRUE

Operation

fd = (cond(cc) == true) ? s1

Instruction Format

op fd, s1

Syntax Example

FDMOV<.cc> fd, s1

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 (s1) are moved to the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (cond(cc) == true) ? s1          /* FDMOV */
```

Assembly Code Example

```
FDMOV fd,s1    ; Move contents of fs1 into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDMOV<.cc>	fd, s1	11100qqq01001000 10QQ0EEEEE1LLLLL

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

(FP_DP_OPTION == TRUE)

Operation

```
fd = s3 – (s2 * s1)
```

Instruction Format

op fd, s3, s2, s1

Syntax Example

FDMSUB fd, s3, s2, s1

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
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 product of s1 and s2 operands is computed, subtracted from s3; the result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)                    /* FDMSUB */
```

Assembly Code Example

```
FDMSUB fd, s3, s1, s2   ; floating-point multiplication of s2 and s1 and the
                        ; product is subtracted from the s3 and result is
                        stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDMSUB	fd, s1, s2, s3	11100mmm011000000MM0EEEEE100110

FDMUL

Function

Double-precision floating-point multiply

Extension Group

FP_DP_OPTION == TRUE

Operation

```
fd = s1 * s2
```

Instruction Format

op fd,s1,s2

Syntax Example

FDMUL fd,s1,s2

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 signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands are multiplied and the product is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* FDMUL */
```

Assembly Code Example

```
FDMUL fd,s1,s2   ; floating-point multiplication of s1 and s2 and the
                 ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDMUL	fd, s1, s2	11100mmm00000010 10MM0EEEEE1LLLLL

FDNMADD

Function

Double-precision floating-point fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding. The negation of the result is stored in the destination register

Extension Group

(FP_DP_OPTION == TRUE)

Operation

```
fd = -(s3 + (s1 * s2))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

FDNMADD fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and a = -infinity or (s2 * s3)= -infinity and a = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands is computed, added to s3, and the negation of 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

```
fd = -(s3 + (s1 * s2))                    /* FDNMADD */
```

Assembly Code Example

```
FDNMADD fd, s3, s1, s2     ; floating-point multiplication of s2 and s3
                           and the product is added to s1 and the negated
                           result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDNMADD	s1, s2, s3	11100mmm00000001 10MM 1EEEEE 0LLLLL

FDNMSUB

Function

Double-precision floating-point fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding. The result is negated and stored in the destination register

Extension Group

(FP_DP_OPTION == TRUE)

Operation

```
fd = -(s3 – (s1 * s2))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

FDNMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s1 * s2)= +infinity and s3 = +infinity or (s1 * s2)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
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 product of s1 and s2 operands is computed, subtracted from s3; the negated result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))                /* FDNMSUB */
```

Assembly Code Example

```
FDNMSUB fd, s3, s2, s1    ; floating-point multiplication of s2 and s1 and the
                          ; product is subtracted from the s3 and result is negated
                          and stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FDNMSUB	fd, s3, s2, s1	11100mmm00000001 10MM1EEEEE1LLLLL

FDSGNJ

Function

Extension Group

FP_DP_OPTION == TRUE

Operation

fd.{s, e, m} = { s2.s, s1.e, s1.m }

Instruction Format

op fd, s1, s2

Syntax Example

FDSGNJ fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { s2.s, s1.e, s1.m }      /* FDSGNJ*/
```

Assembly Code Example

```
FDSGNJ fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FDSGNJ	fd, s1, s2	11100mmm00001000010MM0EEEEE1LLLLL

FDSGNJN

Function

Extension Group

FP_DP_OPTION == TRUE

Operation

fd.{s, e, m} = { xor(s1.s, s2.s), s1.e, s1.m}

Instruction Format

op fd, s1, s2

Syntax Example

FDSGNJN fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { xor(s1.s, s2.s), s1.e, s1.m}   /* FDSGNJN*/
```

Assembly Code Example

```
FDSGNJN fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FDSGNJN	fd, s1, s2	11100mmm00001010 10MM0EEEEE1LLLLL

FDSGNJX

Function

Extension Group

FP_DP_OPTION == TRUE

Operation

fd.{s, e, m} = { xor(s1.s, s2.s), s1.e, s1.m}

Instruction Format

op fd, s1, s2

Syntax Example

FDSGNJX fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { xor(s1.s, s2.s), s1.e, s1.m}   /* FDSGNJX*/
```

Assembly Code Example

```
FDSGNJX fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FDSGNJX	fd, s1, s2	11100mmm01100000000MM0EEEEE100011

FDSQRT

Function

Double-precision floating-point square-root

Extension Group

(FP_DP_OPTION == TRUE) && (FPU_DIV_OPTION == TRUE)

Operation

```
fd = sqrt (s1)
```

Instruction Format

op fd, s1

Syntax Example

FDSQRT fd, s1

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 signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
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 fd operand contains the square-root of the s1 operand.

Pseudo Code

```
fd = sqrt (s1)                          FDSQRT
```

Assembly Code Example

```
FDSQRT fd, s1    fd operand contains the square-root of the s1 operand
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
FDSQRT            fd, s1       11100000010000001000 0EEEEE1LLLLL
```

FDSUB

Function

Double-precision floating-point subtraction

Extension Group

FP_DP_OPTION == TRUE

Operation

```
fd = s1- s2
```

Instruction Format

op fd, s1, s2

Syntax Example

FDSUB fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
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 difference of s1 and s2 operands is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*FDSUB
```

Assembly Code Example

```
FDSUB fd, s1, s2     difference of s1 and s2 operands is stored in fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
FDSUB              fd, s1, s2    11100mmm00000001 10MM0EEEEE1LLLLL
```

FHADD

Function

Half-precision floating-point addition

Extension Group

FP_HP_OPTION == TRUE

Operation

```
fd = s1 +s2
```

Instruction Format

op fd, s1, s2

Syntax Example

FHADD fd, s1, s2

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 signalling NaN when FP_CTRLFP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The sum of s1 and s2 operands is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* FHADD */
```

Assembly Code Example

```
FHADD fd, s1, s2       ; floating-point addition of s1 and s2 and the
                       ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHADD	fd, s1, s2	11100mmm00000000 00MM0EEEEE1LLLLL

FHCMP

Function

Half-precision floating-point comparison

Extension Group

FP_HP_OPTION == TRUE

Operation

```
ZNCV <- FHCMP(s1, s2)
```

Instruction Format

op s1, s2

Syntax Example

FHCMP s1,s2

STATUS32 Flags Affected

Z	•	= Set when s1 and s2 are equal
N	•	= Set when s1 is less than s2
C	•	= Set when s1 is less than s2
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 FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The s1 and s2 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

```
ZNCV <- FHCMP(s1, s2)                                                    /* FHCMP */
```

Assembly Code Example

```
FHCMP s1,s2       ; floating-point comparison of s1 and s2 and the
                  ; core flags are updated
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHCMP	s1,s2	11100mmm000001000MM0EEEEE1LLLLL

FHCMPF

Function

Half-precision floating-point comparison operation with IEEE 754 flag generation. This instruction is similar to the FHCMP instruction in cases when either of the instruction operands is a signaling NaN. The FHCMPF instruction updates the invalid flag (FPU_STATUS.IV) when either of the operands is a quiet or signaling NaN, whereas, the FHCMP instruction updates the invalid flag (FPU_STATUS.IV) only when either of the operands is a signaling NaN.

Extension Group

FP_HP_OPTION == TRUE

Operation

```
ZNCV <- FHCMP(s1, s2)    "signaling comparison"
```

Instruction Format

op s1, s2

Syntax Example

FHCMPF s1, s2

STATUS32 Flags Affected

Z	•	= Set when s1 and s2 are equal
N	•	= Set when s1 is less than s2
C	•	= Set when s1 is less than s2
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 FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The s1 and s2 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

```
ZNCV <- FHCMP(s1, s2)    "signaling comparison"            /* FHCMPF */
```

Assembly Code Example

```
FHCMPF s1,s2     ; floating-point comparison of s1 and s2 and update flags
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHCMPF	s1, s2	11100mmm000001010 0MM0EEEEE1LLLLL

FHDIV

Function

Half-precision floating-point division

Extension Group

(FP_HP_OPTION == TRUE) && (FPU_DIV_OPTION == TRUE)

Operation

```
fd = s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

FSDIV fd, s1, s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ	•	Set when the divisor is zero
OF	•	Set when the normalized result exceeds the maximumrepresentable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The s1 operand is divided by the s2 operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                                /* FHDIV */
```

Assembly Code Example

```
FHDIV fd,s1,s2      ; floating-point division of s1 by s2 and the
                    ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHDIV	fd, s1, s2	11100mmm00000011100MM0EEEEE1LLLLL

FHMADD

Function

Half-precision floating-point fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding.

Extension Group

(FP_HP_OPTION == TRUE)

Operation

```
fd = s3 + (s1 * s2)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

FHMADD fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s3, s1, and s2 operands are in the following combinations: (s2 * s1)= +infinity and s3 = -infinity or (s2 * s1)= -infinity and s3 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded after the accumulation

Description

The product of s2 and s1 operands is computed, added to s3, 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

```
fd = s3 + (s1 * s2)                          /* FHMADD */
```

Assembly Code Example

```
FHMADD fd, s3, s1, s3      ; floating-point multiplication of s2 and s1
                           and the ; product is accumulated and the sum
                           stored in s3.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHMADD	s3, s1, s2	11100mmm00000000000MM1EEEEE0LLLLL

FHMAX

Function

Return maximum value of two half-precision floating-point numbers

Extension Group

FP_HP_OPTION == TRUE

Operation

fd = (s1 > s2) ? s1 : s2

Instruction Format

op fd, s1, s2

Syntax Example

FHMAX fd, s1, s2

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 floating-point operands (s1 and s2) and place the result in the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (s1 > s2) ? s2 : s1                /* FHMAX */
```

Assembly Code Example

```
FHMAX fd, s1, s2    ; Take maximum of s1 and s2 and write result into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHMAX	fd, s1, s2	11100mmm00000111100MM0EEEEE1LLLLL

FHMIN

Function

Return the minimum value of two half-precision floating-point numbers

Extension Group

FP_HP_OPTION == TRUE

Operation

fd = (s1 > s2) ? s2 : s1

Instruction Format

op fd, s1, s2

Syntax Example

FHMIN<.f> fd, s1, s2

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 floating-point operands (s1 and s2) and place the result in the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (s1 > s2) ? s2 : s1                /* FHMIN */
```

Assembly Code Example

```
FHMIN fd, s1, s2    ; Take minimum of s1 and s2 and write result into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHMIN	fd, s1, s2	1s1100mmm000001100 0MM0EEEEE1LLLLL

FHMOVCC

Function

Move contents from the half-precision source register to the destination register if the conditions are met

Extension Group

FP_HP_OPTION == TRUE

Operation

fd = (cond(cc) == true) ? s1

Instruction Format

op fd, s1

Syntax Example

FHMOV<.cc> fd, s1

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 (s1) are moved to the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (cond(cc) == true) ? s1          /* FHMOV */
```

Assembly Code Example

```
FHMOV fd,s1                  ; Move contents of fs1 into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHMOV<.cc>	fd, s1	11100qqq01001000000QQ0EEEEE1LLLLL

FHMSUB

Function

Half-precision floating-point fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding.

Extension Group

(FP_HP_OPTION == TRUE)

Operation

```
fd = s3 – (s1 * s2)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

FHMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The product of s1 and s2 operands is computed, subtracted from s3; the result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)                    /* FHMSUB */
```

Assembly Code Example

```
FHMSUB fd, s3, s1, s2     The product of s1 and s2 operands is computed,
                          subtracted from s3; the result is rounded and
                          stored in the destination register.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHMSUB	fd, s3, s1, s2	11100mmm00000000000MM1EEEEE1LLLLL

FHMUL

Function

Half-precision floating-point multiply

Extension Group

FP_HP_OPTION == TRUE

Operation

```
fd = s1 * s2
```

Instruction Format

op fd,s1,s2

Syntax Example

FHMUL fd,s1,s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		= Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The s1 and s2 operands are multiplied and the product is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* FHMUL */
```

Assembly Code Example

```
FHMUL fd,s1,s2   ; floating-point multiplication of s1 and s2 and the
                 ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHMUL	fd, s1, s2	11100mmm000000100 0MM0EEEEE1LLLLL

FHNMADD

Function

Half-precision floating-point fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding. The negation of the result is stored in the destination register

Extension Group

(FP_HP_OPTION == TRUE)

Operation

```
fd = -(s3 + (s1 * s2))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

FHNMADD fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and a = -infinity or (s2 * s3)= -infinity and a = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands is computed, added to s3, and the negation of 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

```
fd = -(s3 + (s1 * s2))                    /* FHNMADD */
```

Assembly Code Example

```
FHNMADD fd, s3, s1, s2     ; floating-point multiplication of s2 and s3
                           and the product is added to s1 and the negated
                           result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHNMADD	s1, s2, s3	11100mmm00000001 00MM1EEEEE0LLLLL

FHNMSUB

Function

Half-precision floating-point fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding. The result is negated and stored in the destination register

Extension Group

(FP_HP_OPTION == TRUE)

Operation

```
fd = -(s3 – (s1 * s2))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

FHNMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s1 * s2)= +infinity and s3 = +infinity or (s1 * s2)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The product of s1 and s2 operands is computed, subtracted from s3; the negated result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))                /* FHNMSUB */
```

Assembly Code Example

```
FHNMSUB fd, s3, s2, s1     ; floating-point multiplication of s2 and s1 and the
                           ; product is subtracted from the s3 and result is negated
                           and stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FHNMSUB	fd, s3, s2, s1	11100mmm00000001 00MM1EEEEE1LLLLL

FHSGNJ

Function

Extension Group

FP_HP_OPTION == TRUE

Operation

fd.{s, e, m} = { s2.s, s1.e, s1.m }

Instruction Format

op fd, s1, s2

Syntax Example

FHSGNJ fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { s2.s, s1.e, s1.m }      /* FHSGNJ*/
```

Assembly Code Example

```
FHSGNJ fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FHSGNJ	fd, s1, s2	11100mmm00001000000MM0EEEEE1LLLLL

FHSGNJN

Function

Extension Group

FP_HP_OPTION == TRUE

Operation

fd.{s, e, m} = { not(s2.s), s1.e, s1.m }

Instruction Format

op fd, s1, s2

Syntax Example

FHSGNJN fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { not(s2.s), s1.e, s1.m }    /* FHSGNJN*/
```

Assembly Code Example

```
FHSGNJN fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FHSGNJN	fd, s1, s2	11100mmm00001010000MM0EEEEE1LLLLL

FHSGNJX

Function

Extension Group

FP_HP_OPTION == TRUE

Operation

fd.{s, e, m} = { xor(s1.s, s2.s), s1.e, s1.m}

Instruction Format

op fd, s1, s2

Syntax Example

FHSGNJX fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { xor(s1.s, s2.s), s1.e, s1.m}   /* FHSGNJX*/
```

Assembly Code Example

```
FHSGNJX fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FHSGNJX	fd, s1, s2	11100bbb111000000BB0EEEEE100010

FHSQRT

Function

Half-precision floating-point square-root

Extension Group

(FP_HP_OPTION == TRUE) && (FPU_DIV_OPTION == TRUE)

Operation

```
fd = sqrt (s1)
```

Instruction Format

op fd, s1

Syntax Example

FHSQRT fd, s1

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		= Unchanged
OF		= Unchanged
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The fd operand contains the square-root of the s1 operand.

Pseudo Code

```
fd = sqrt (s1)                          FHSQRT
```

Assembly Code Example

```
FHSQRT fd, s1    fd operand contains the square-root of the s1 operand
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
FHSQRT            fd, s1       11100000010000000000EEEEE1LLLLL
```

FHSUB

Function

Half-precision floating-point subtraction

Extension Group

FP_HP_OPTION == TRUE

Operation

```
fd = s1- s2
```

Instruction Format

op fd, s1, s2

Syntax Example

FHSUB fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The difference of s1 and s2 operands is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*FHSUB
```

Assembly Code Example

```
FHSUB fd, s1, s2     difference of s1 and s2 operands is stored in fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
FHSUB              fd, s1, s2    11100mmm00000001 00MM0EEEEE1LLLLL
```

FLDD32

Function

Loads the values from the memory at the specified address to an even-numbered pair of floating-point destination registers.

Extension Group

ARC64: Baseline HAS_FP == TRUE

Operation

```
FLDD32 fd, [EA]{ FLD32 fd, [EA] ;     FLD32 fd+1, [EA+4] }
```

Instruction Format

op `fd,[b,s9]`

Syntax Example

FLDD32 `fd,[b,s9]`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Loads the values from the memory at the specified address to an even-numbered pair of floating-point destination registers. If `fpr_width` is `64`, an FLDD32 instruction clears the upper 32 bits of fd and fd+1.

Specifying an odd-numbered register as an operand to this instruction raises an illegal instruction exception.

Pseudo Code

```
                                        /* FLDD32*/
```

Assembly Code Example

```
FLDD32 fd,[b,s9]          ; Loads the values from the memory at the
                          specified address to an even-numbered pair of
                          floating-point destination registers.
```

Syntax and Encoding

		Instruction Code
FLDD32	fd, [b,s9]	01101bbbssssssssSBBB0EEEEE1aa000

FLDD64

Function

Loads two consecutive 64-bit values from the specified memory address.

Extension Group

HAS_FP == TRUE

Operation

Instruction Format

op `fd,[b,s9]`

Syntax Example

FLDD64 `fd,[b,s9]`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Loads two consecutive 64-bit values from the specified memory address and places element 0 in the named destination floating-point register. Element 1 is placed in the next sequential floating-point register.

Pseudo Code

```
                                        /* FLDD64*/
```

Assembly Code Example

```
FLDD64 f0,[b,s9]          ; load a 128-bit value from the specified memory
                          address
```

Syntax and Encoding

		Instruction Code
FLDD64	fd,[b,s9]	01101bbbssssssssSBBB0EEEEE1aa010

FLD16

Function

Loads a 16-bit value from the specified memory address and places it into the lower 16 bits of the destination floating-point register. All bits above bit 15 in the destination floating-point register are cleared.

Extension Group

HAS_FP == TRUE

Operation

Instruction Format

op `fd,[b,s9]`

Syntax Example

FLD16 `fd,[b,s9]`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Loads a 16-bit value from the specified memory address and places it into the lower 16 bits of the destination floating-point register. All bits above bit 15 in the destination floating-point register are cleared.

Pseudo Code

```
                                        /* FDL16*/
```

Assembly Code Example

```
FLD16 f0,[b,s9]            ; load a 16-bit value from the specified memory
                           address and places it into the lower 16-bits of
                           the destination floating-point register.
```

Syntax and Encoding

		Instruction Code
FLD16	fd,[b,s9]	01101bbbssssssssSBBB0EEEEE0aa100

FLD32

Function

Loads a 32-bit value from the specified memory address and places it into the lower 32 bits of the destination floating-point register. If double-precision floating-point unit is configured, then all bits above bit 31 in the destination floating-point register are cleared.

Extension Group

FP_DP_OPTION == TRUE

Operation

Instruction Format

op `fd,[b,s9]`

Syntax Example

FLD32 `fd,[b,s9]`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Loads a 32-bit value from the specified memory address and places it into the lower 32 bits of the destination floating-point register. If double-precision floating-point unit is configured, then all bits above bit 31 in the destination floating-point register are cleared.

Pseudo Code

```
                                        /* FDL32*/
```

Assembly Code Example

```
FLD32 f0,[b,s9]            ; load a 32-bit value from the specified memory
                           address and places it into the lower 32-bits of
                           the destination floating-point register.
```

Syntax and Encoding

		Instruction Code
FLD32	fd,[b,s9]	01101bbbssssssssSBBB0EEEEE0aa000

FLD64

Function

Loads a 64-bit value from a double-precision register to the specified memory address.

Extension Group

FP_DP_OPTION == TRUE

Operation

Instruction Format

op `fd,[b,s9]`

Syntax Example

FLD64 `fd,[b,s9]`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Loads a 64-bit value from a double-precision register to the specified memory address. Any attempt to execute these instructions when -fp_dp_option is false will raise an Illegal Instruction exception

Pseudo Code

```
                                        /* FLD64*/
```

Assembly Code Example

```
FLD64 f0,[b,s9]             ; load a 64-bit value from the specified memory
                            address
```

Syntax and Encoding

		Instruction Code
FLD64	fd,[b,s9]	01101bbbssssssssSBBB0EEEEE0aa010

FMVD2L

Function

Move contents of a floating-point register to a general-purpose register without type conversion

Extension Group

HAS_FP == TRUE

Operation

```
if cc==true then
dest = src
```

Instruction Format

op b, e, u5

Syntax Example

FMVD2L b, e, u5

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction moves contents of a floating-point register to a general-purpose register. The following table summarizes the behavior of this instruction.

ISA	-fp_dp_option	-fp_vec_option && -fp_wide_option	Instruction	Semantics	Bits moved	GPRs used	FPRs used
ARC64	False	False	FMVD2L	Raises Illegal Instruction	-	-	-
		True	FMVD2L	Rd ← { Fs+1, Fs }	64	1	2
	True	X	FMVD2L	Rd ← Fs	64	1	1

Pseudo Code

```
if cc==true then                          /* FMVD2L*/
 dest = src
```

Assembly Code Example

```
FMVD2L r1,f0, u5   ; move from f0 to r1
```

Syntax and Encoding

		Instruction Code
FMVD2L	b,e,u5	11100mmm011000**11**00MM0BBBBB**1**10001

FMVI2S

Function

Move contents of a general-purpose register to a single-precision floating-point register without type conversion

Extension Group

HAS_FP == TRUE

Operation

```
if cc==true then
    dest = src
```

Instruction Format

op e, b, u5

Syntax Example

FMVI2s e, b, u5

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction moves contents of a general-purpose register to a single-precision floating-point register. The following table summarizes the behavior of this instruction.

ISA	-fp_dp_option	-fp_vec_option && -fp_wide_option	Instruction	Semantics	Bits moved	GPRs used	FPRs used
ARC64	False	X	FMVI2S	Fd ← Rs [31:0]	32	1	1
	True	X	FMVI2S	Fd ← { 0x0000_0000, Rs [31:0] }	32	1	1

Pseudo Code

```
if cc==true then                          /* FMVI2S*/
 dest = src
```

Assembly Code Example

```
FMVI2s f0,r1, u5   ; move from r1 to f0
```

Syntax and Encoding

		Instruction Code
FMVI2S<.cc>	e,b,u5	11100bbb111000000BB0EEEEE110000

FMVL2D

Function

Move contents of a general-purpose register to a floating-point register without type conversion

Extension Group

HAS_FP == TRUE

Operation

```
if cc==true then
dest = src
```

Instruction Format

op e, b, u5

Syntax Example

FMVL2D b, e, u5

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction moves contents of a general-purpose register to a floating-point register. The following table summarizes the behavior of this instruction.

ISA	-fp_dp_option	-fp_vec_option && -fp_wide_option	Instruction	Semantics	Bits moved	GPRs used	FPRs used
ARC64	False	False	FMVL2D	Raises Illegal Instruction	-	-	-
		True	FMVL2D	{ Fd+1, Fd } ← Rs	64	1	2
	True	X	FMVL2D	Fd ← Rs	64	1	1

Pseudo Code

```
if cc==true then                          /* FMVL2D*/
 dest = src
```

Assembly Code Example

```
FMVL2D f0,r1, u5   ; move from r1 to f0
```

Syntax and Encoding

		Instruction Code
FMVL2D	e,b,u5	11100bbb111000**11**00BB0EEEEE110000

FMVS2I

Function

Move contents of a floating-point register to a general-purpose register without type conversion

Extension Group

HAS_FP == TRUE

Operation

```
if cc==true then
dest = src
```

Instruction Format

op b, e, u5

Syntax Example

FMVS2I b, e, u5

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction moves contents of a floating-point register to a general-purpose register. The following table summarizes the behavior of this instruction.

ISA	-fp_dp_option	-fp_vec_option && -fp_wide_option	Instruction	Semantics	Bits moved	GPRs used	FPRs used
ARC64	False	X	FMVS2I	Rd ← { 0x0000_0000, Fs }	32	1	1
	True	X	FMVS2I	Rd ← { 0x0000_0000, Fs [31:0] }	32	1	1

Pseudo Code

```
if cc==true then                          /* FMVS2I*/
 dest = src
```

Assembly Code Example

```
FMVS2I r1,f0, u5   ; move from f0 to r1
```

Syntax and Encoding

		Instruction Code
FMVS2I	b,e,u5	11100eee011000000EE0BBBBB110001

FSADD

Function

Single-precision floating-point addition

Extension Group

HAS_FP == TRUE

Operation

```
fd = s1 +s2
```

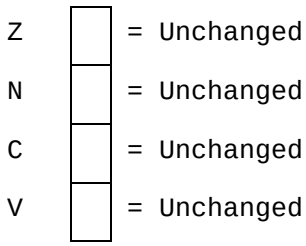
Instruction Format

op fd, s1, s2

Syntax Example

FSADD fd, s1, s2

STATUS32 Flags Affected



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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* FSADD */
```

Assembly Code Example

```
FSADD fd, s1, s2   ; floating-point addition of s1 and s2 and the
                   ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSADD	fd, s1, s2	11100mmm00000000101MM0EEEEE1LLLLL

FSCMP

Function

Single-precision floating-point comparison

Extension Group

HAS_FP == TRUE

Operation

```
ZNCV <- FSCMP(s1, s2)
```

Instruction Format

op s1,s2

Syntax Example

FSCMP s1,s2

STATUS32 Flags Affected

Z	•	= Set when s1 and s2 are equal
N	•	= Set when s1 is less than s2
C	•	= Set when s1 is less than s2
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 FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The s1 and s2 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

```
ZNCV <- FSCMP(s1, s2)                                                    /* FSCMP */
```

Assembly Code Example

```
FSCMP s1,s2       ; floating-point comparison of s1 and s2 and the
                  ; core flags are updated
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSCMP	s1,s2	11100mmm000001000 1MM0EEEEE1LLLLL

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

HAS_FP == TRUE

Operation

```
ZNCV <- FSCMP(s1, s2)    "signaling comparison"
```

Instruction Format

op s1, s2

Syntax Example

FSCMPF s1, s2

STATUS32 Flags Affected

Z	•	= Set when s1 and s2 are equal
N	•	= Set when s1 is less than s2
C	•	= Set when s1 is less than s2
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 FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF		Unchanged
UF		Unchanged
IX		Unchanged

Description

The s1 and s2 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

```
ZNCV <- FSCMP(s1, s2)    "signaling comparison"            /* FSCMPF */
```

Assembly Code Example

```
FSCMPF s1,s2     ; floating-point comparison of s1 and s2 and update flags
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSCMPF	s1, s2	11100mmm000001000 1MM0EEEEE1LLLLL

FSDIV

Function

Single-precision floating-point division

Extension Group

HAS_FP == TRUE && FPU_DIV_OPTION == TRUE

Operation

```
fd = s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

FSDIV fd, s1, s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 operand is divided by the s2 operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                              /* FSDIV */
```

Assembly Code Example

```
FSDIV fd, s1, s2      ; floating-point division of s1 by s2 and the
                      ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSDIV	fd, s1, s2	11100mmm00000011101MM0EEEEE1LLLLL

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_FP == TRUE)

Operation

```
fd = s3 + (s1 * s2)
```

Instruction Format

op fd, s1, s2, s3

Syntax Example

FSMADD fd, s1, s2, s3

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	▪ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ▪ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s3)= +infinity and s1 = -infinity or (s2 * s3)= -infinity and s1 = +infinity ▪ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s2 and s3 operands is computed, added to s1, 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

```
fd = s1 + (s2 * s3)                    /* FSMADD */
```

Assembly Code Example

```
FSMADD fd, s1, s2, s3      ; floating-point multiplication of s2 and s3
                           and the ; product is accumulated and the sum
                           stored in s1.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSMADD	s1, s2, s3	11100mmm00000110 10MM 1EEEEE 0LLLLL

FSMAX

Function

Return maximum value of two single-precision floating-point numbers

Extension Group

HAS_FP == TRUE

Operation

fd = (s1 > s2) ? s1 : s2

Instruction Format

op fd, s1, s2

Syntax Example

FSMAX fd, s1, s2

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 floating-point operands (s1 and s2) and place the result in the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (s1 > s2) ? s2 : s1                /* FSMAX */
```

Assembly Code Example

```
FSMAX fd, s1, s2    ; Take maximum of s1 and s2 and write result into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSMAX	fd, s1, s2	11100mmm00000111101MM0EEEEE1LLLLL

FSMIN

Function

Return minimum value of two single-precision floating-point numbers

Extension Group

HAS_FP == TRUE

Operation

fd = (s1 > s2) ? s2 : s1

Instruction Format

op fd, s1, s2

Syntax Example

FSMIN<.f> fd, s1, s2

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 floating-point operands (s1 and s2) and place the result in the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (s1 > s2) ? s2 : s1                /* FSMIN */
```

Assembly Code Example

```
FSMIN fd, s1, s2    ; Take minimum of s1 and s2 and write result into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSMIN	fd, s1, s2	11100mmm00001100 1MM0EEEEE1LLLLL

FSMOVCC

Function

Move contents from the single-precision source register to the destination register if the conditions are met

Extension Group

HAS_FP == TRUE

Operation

fd = (cond(cc) == true) ? s1

Instruction Format

op fd, s1

Syntax Example

FSMOV<.cc> fd, s1

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 (s1) are moved to the destination register (fd).

Any flag updates occur only if the set flags suffix (.F) is used.

Pseudo Code

```
fd = (cond(cc) == true) ? s1          /* FSMOV */
```

Assembly Code Example

```
FSMOV fd,s1    ; Move contents of fs1 into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSMOV<.cc>	fd, s1	11100qqq01001000001QQ0EEEEE1LLLLL

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_FP == TRUE)

Operation

```
fd = s3 – (s2 * s1)
```

Instruction Format

op fd, s3, s2, s1

Syntax Example

FSMSUB fd, s3, s2, s1

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands is computed, subtracted from s3; the result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)                    /* FSMSUB */
```

Assembly Code Example

```
FSMSUB fd, s3, s1, s2   ; floating-point multiplication of s2 and s1 and the
                        ; product is subtracted from the s3 and result is
                        stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSMSUB	fd, s1, s2, s3	11100mmm00000000 01MM 1EEEEE 1LLLLL

FSMUL

Function

Single-precision floating-point multiply

Extension Group

HAS_FP == TRUE

Operation

```
fd = s1 * s2
```

Instruction Format

op fd,s1,s2

Syntax Example

FSMUL fd,s1,s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands are multiplied and the product is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* FSMUL */
```

Assembly Code Example

```
FSMUL fd,s1,s2   ; floating-point multiplication of s1 and s2 and the
                 ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSMUL	fd, s1, s2	11100mmm000000100 1MM0EEEEE1LLLLL

FSNMADD

Function

Single-precision floating-point fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding. The negation of the result is stored in the destination register

Extension Group

(HAS_FP == TRUE)

Operation

```
fd = -(s3 + (s1 * s2))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

FSNMADD fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and a = -infinity or (s2 * s3)= -infinity and a = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands is computed, added to s3, and the negation of 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

```
fd = -(s3 + (s1 * s2))                    /* FSNMADD */
```

Assembly Code Example

```
FSNMADD fd, s3, s1, s2     ; floating-point multiplication of s2 and s3
                           and the product is added to s1 and the negated
                           result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSNMADD	s1, s2, s3	11100mmm00000010 1MM1EEEEE0LLLLL

FSNMSUB

Function

Single-precision floating-point fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding. The result is negated and stored in the destination register

Extension Group

(HAS_FP == TRUE)

Operation

```
fd = -(s3 – (s1 * s2))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

FSNMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s1 * s2)= +infinity and s3 = +infinity or (s1 * s2)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands is computed, subtracted from s3; the negated result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))                /* FSNMSUB */
```

Assembly Code Example

```
FSNMSUB fd, s3, s2, s1      ; floating-point multiplication of s2 and s1 and the
                            ; product is subtracted from the s3 and result is negated
                            and stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
FSNMSUB	fd, s3, s2, s1	11100mmm00000001 01MM1EEEEE1LLLLL

FSSGNJ

Function

Extension Group

HAS_FP == TRUE

Operation

fd.{s, e, m} = { s2.s, s1.e, s1.m }

Instruction Format

op fd, s1, s2

Syntax Example

FSSGNJ fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { s2.s, s1.e, s1.m }      /* FSSGNJ*/
```

Assembly Code Example

```
FSSGNJ fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FSSGNJ	fd, s1, s2	11100mmm00001000001MM0EEEEE1LLLLL

FSSGNJN

Function

Extension Group

HAS_FP == TRUE

Operation

fd.{s, e, m} = { not(s2.s), s1.e, s1.m }

Instruction Format

op fd, s1, s2

Syntax Example

FSSGNJN fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { not(s2.s), s1.e, s1.m }   /* FSSGNJN*/
```

Assembly Code Example

```
FSSGNJN fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FSSGNJN	fd, s1, s2	11100mmm00001010001MM0EEEEE1LLLLL

FSSGNJX

Function

Extension Group

HAS_FP == TRUE

Operation

fd.{s, e, m} = { xor(s1.s, s2.s), s1.e, s1.m}

Instruction Format

op fd, s1, s2

Syntax Example

FSSGNJX fd, s1, s2

STATUS32 Flags Affected

Z	•
N	•
C	•
V	•

Description

Pseudo Code

```
fd.{s, e, m} = { xor(s1.s, s2.s), s1.e, s1.m}   /* FSSGNJX*/
```

Assembly Code Example

```
FSSGNJX fd, s1, s2
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

Instruction Code

FSSGNJX	fd, s1, s2	11100bbb111000000BB0EEEEE100010

FSSQRT

Function

Single-precision floating-point square-root

Extension Group

(HAS_FP == TRUE) && (FPU_DIV_OPTION == TRUE)

Operation

```
fd = sqrt (s1)
```

Instruction Format

op fd, s1

Syntax Example

FSSQRT fd, s1

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 fd operand contains the square-root of the s1 operand.

Pseudo Code

```
fd = sqrt (s1)                          FSSQRT
```

Assembly Code Example

```
FSSQRT fd, s1    fd operand contains the square-root of the s1 operand
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
FSSQRT            fd, s1       11100000010000000100EEEEE1LLLLL
```

FSSUB

Function

Single-precision floating-point subtraction

Extension Group

HAS_FP == TRUE

Operation

```
fd = s1-s2
```

Instruction Format

op fd, s1, s2

Syntax Example

FSSUB fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 s1 and s2 operands is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*FSSUB
```

Assembly Code Example

```
FSSUB fd, s1, s2     difference of s1 and s2 operands is stored in fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

FSSUB	fd, s1, s2	11100mmm00000001 01MM0EEEEE1LLLLL

FSTD64

Function

Stores two 64-bit values from an even-numbered pair of double-precision floating-point source registers, to memory at the specified address. The even-numbered floating-point register is stored to the lower 64-bit address and the odd-numbered floating-point register is stored to the higher 64-bit address.

Extension Group

FP_DP_OPTION == TRUE

Operation

Instruction Format

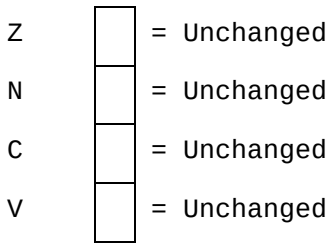
op `[b,s9], fs`

Syntax Example

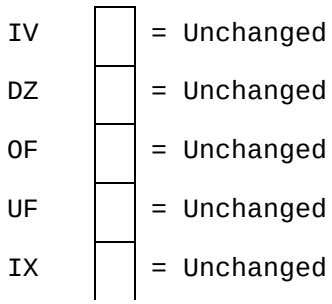
FSTD64 `[b,s9], fs`

STATUS32 Flags Affected

STATUS32 Flags



FPU_STATUS Flags



Description

Stores two 64-bit values from an even-numbered pair of double-precision floating-point source registers, to memory at the specified address. The even-numbered floating-point register is stored to the lower 64-bit address and the odd-numbered floating-point register is stored to the higher 64-bit address.

Pseudo Code

```
                                        /* FSTD64*/
```

Assembly Code Example

```
FSTD64 [b,s9], fs          ; stores two 64-bit values from an even-numbered
                           pair of double-precision floating-point source
                           registers, to memory at the specified address.
                           The even-numbered floating-point register is
                           stored to the lower 64-bit address and the odd-
                           numbered floating-point register is stored to
                           the higher 64-bit address.
```

Syntax and Encoding

		Instruction Code
FSTD64	[b,s9], fs	01101bbbssssssssSBBB0EEEEE1aa011

FST16

Function

Stores a 16-bit value from the lower 16 bits of the designated floating-point source register to memory, at the specified address

Extension Group

HAS_FP == TRUE

Operation

Instruction Format

op `[b,s9], fs`

Syntax Example

FST16 `[b,s9], fs`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Stores a 16-bit value from the lower 16 bits of the designated floating-point source register to memory, at the specified address

Pseudo Code

```
                                        /* FST16*/
```

Assembly Code Example

```
FST16 [b,s9], fs          ; Stores a 16-bit value from the lower 16 bits
                          of the designated floating-point source
                          register to memory
```

Syntax and Encoding

		Instruction Code
FST16	[b,s9], fs	01101bbbssssssssSBBB0EEEEE0aa101

FST32

Function

Stores a 32-bit value from the lower 32 bits of the designated floating-point source register to memory, at the specified address

Extension Group

HAS_FP == TRUE

Operation

Instruction Format

op `[b,s9], fs`

Syntax Example

FST32 `[b,s9], fs`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Stores a 32-bit value from the lower 32 bits of the designated floating-point source register to memory, at the specified address

Pseudo Code

```
                                        /* FST32*/
```

Assembly Code Example

```
FST32 [b,s9], fs          ; Stores a 32-bit value from the lower 32 bits
                          of the designated floating-point source
                          register to memory, at the specified address
```

Syntax and Encoding

		Instruction Code
FST32	[b,s9], fs	01101bbbssssssssSBBB0EEEEE0aa001

FST64

Function

Stores a 64-bit value from one double-precision floating-point source register to memory at the specified address.

Extension Group

FP_DP_OPTION == TRUE

Operation

Instruction Format

op `[b,s9], fs`

Syntax Example

FST64 `[b,s9], fs`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Stores a 64-bit value from one double-precision floating-point source register to memory at the specified address. Any attempt to execute these instructions when -fp_dp_option is false will raise an Illegal Instruction exception.

Pseudo Code

```
                                        /* FST64*/
```

Assembly Code Example

```
FST64 [b,s9], fs          ; stores a 64-bit value from one double-
                          precision floating-point source register to
                          memory at the specified address.
```

Syntax and Encoding

		Instruction Code
FST64	[b,s9], fs	01101bbbssssssssSBBB0EEEEE0aa011

FSTD32

Function

Stores the values from an even-numbered pair of floating-point source registers to the memory at the specified address.

Extension Group

ARC64: Baseline HAS_FP == TRUE

Operation

```
FSTD32 fs1, [EA]{ FST32 fs1, [EA] ;    FST32 fs1+1, [EA+4] }
```

Instruction Format

op `[b,s9], fs`

Syntax Example

FSTD32 `[b,s9], fs`

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

Stores the values from an even-numbered pair of floating-point source registers to the memory at the specified address. If `fpr_width` is `64`, an FSTD32 instruction ignores the upper 32 bits of fs1 and fs1+1,

and therefore does not contribute to the memory write data. Specifying an odd-numbered register as an operand to this instruction raises an illegal instruction exception.

Pseudo Code

```
                                        /* FSTD32*/
```

Assembly Code Example

```
FSTD32 [b,s9], fs          ; stores a values from an even-numbered pair of
                           floating-point source registers, to memory at
                           the specified address.
```

Syntax and Encoding

		Instruction Code
FSTD32	[b,s9], fs	01101bbbssssssssSBBB0EEEEE1aa001

VFDADD

Function

Half-precision floating-point vector addition

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 + s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFDADD fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 double-precision elements in a vector are added with the corresponding double-precision elements in the second vector. The sum is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* VFDADD */
```

Assembly Code Example

```
VFDADD fd, fs1, fs2   ; vector floating-point addition of s1 and s2
                      and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDADD	fd, s1, s2	11100mmm00010000010MM0EEEEE1LLLLL

VFDADDS

Function

Add a scalar to a vector with double-precision floating-point elements

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 + s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFDADDS fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 scalar value is added with each double-precision element in a vector. The sum is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* VFDADDS */
```

Assembly Code Example

```
VFDADDS fd, s1, s2      ; vector floating-point addition of s1 and
                        scalar s2 and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDADDS	fd, s1, s2	11100mmm00010100 10MM0EEEEE1LLLLL

VFDADDSUB

Function

Instruction performs addition in all even-numbered lanes and subtraction in all odd-numbered lanes.

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

On even-numbered lanes:

```
fd = s1 + s2
```

On odd-numbered elements:

```
fd = s1 - s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFDADDSUB fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Instruction performs addition in all even-numbered lanes and subtraction in all odd-numbered lanes.

Pseudo Code

```
even-numbered lanes:                          /* VFDADDSUB */
  fd = s1 +s2
odd-numbered lanes:
  fd = s1 - s2
```

Assembly Code Example

```
VFDADDSUB fd, s1, s2    ;Instruction performs addition in all even-
                        numbered lanes and subtraction in all odd-
                        numbered lanes.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDADDSUB	fd, s1, s2	11100mmm00001110 10MM0EEEEE1LLLLL

VFDDIV

Function

Double-precision vector floating-point division

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFDDIV fd, s1, s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ	•	Set when the divisor is zero
OF	•	Set when the normalized result exceeds the maximumrepresentable 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 half-precision elements in a vector are divided by the half-precision elements in the second vector operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                                /* VFDDIV */
```

Assembly Code Example

```
VFDDIV fd, s1, s2     ; floating-point division of s1 by s2 and the
                      ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDDIV	fd, s1, s2	11100mmm00010011 10MM0EEEEE1LLLLL

VFDDIVS

Function

Divide each double-precision floating-point vector element by a scalar

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFDDIVS fd, s1, s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Each double-precision element in a vector is divided by a scalar operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                                /* VFDDIVS */
```

Assembly Code Example

```
VFDDIVS fd, s1, s2    ; floating-point division of a vector s1 by a
                      scalar s2 and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDDIVS	fd, s1, s2	11100mmm00010111 10MM0EEEEE1LLLLL

VFDEXT

Function

Extract double-precision element from vector at a literal index or variable index

Extension Group

FP_DP_OPTION == TRUE

Operation

Instruction Format

op fd, fs1 [u5] //extract from literal index

op fd, fs1 [b]//extract from variable index

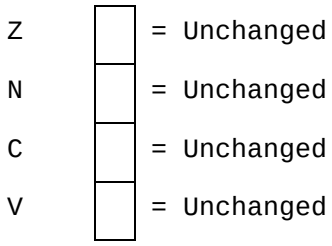
Syntax Example

VDHEXT fd, fs1 [u5]

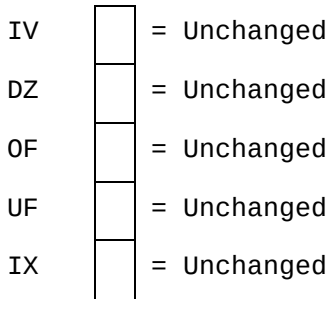
VDHEXT fd, fs1 [b]

STATUS32 Flags Affected

STATUS32 Flags



FPU_STATUS Flags



Description

This instruction extracts double-precision element from vector at a literal index or variable index.

The number of consecutive vector elements contained in a floating-point register is a function VLEN, which depends on VFP_WIDTH and the size of the floating-point type given by the P operand. That is, VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P).

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, each source or destination vector operand for these vector insert/extract/replicate operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

The VF<P>EXT instructions silently ignore any bits of the u5 or B-register index operand that are not needed to index into the vector operand. Effectively, the index operand is always taken modulo VLEN before being used as the index.

It is not illegal to perform extract operations when VLEN(VFP_WIDTH, P) = 1.

Pseudo Code

```
                                        /* VFDEXT*/
```

Assembly Code Example

```
VFDEXT fd, fs1 [u5]        ; extract scalar from a vector at literal index

VFDEXT fd, fs1 [b]         ; extract scalar from a vector at variable index
```

Syntax and Encoding

		Instruction Code
VFDEXT	fd, fs1 [u5]	11100uuu0101000010UU0EEEEE1LLLLL
VFDEXT	fd, fs1 [b]	11100bbb1000000010BB0EEEEE1LLLLL

VFDINS

Function

Insert double-precision element into vector at a literal index or variable index

Extension Group

FP_DP_OPTION == TRUE

Operation

Instruction Format

op fd[u5], fs1 //extract from literal index

op fd [b], fs1//extract from variable index

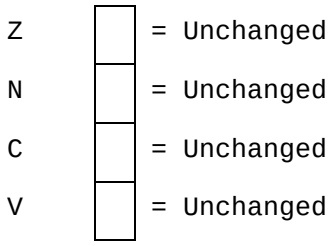
Syntax Example

VFDINS fd[u5], fs1

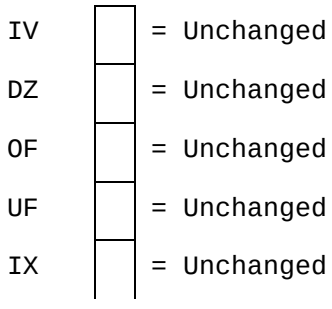
VFDINS fd [b], fs1

STATUS32 Flags Affected

STATUS32 Flags



FPU_STATUS Flags



Description

This instruction inserts double-precision element into vector at a literal index or variable index.

The number of consecutive vector elements contained in a floating-point register is a function VLEN, which depends on VFP_WIDTH and the size of the floating-point type given by the P operand. That is, VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P).

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, each source or destination vector operand for these vector insert operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

The VF<P>INS instructions silently ignore any bits of the u5 or B-register index operand that are not needed to index into the vector operand. Effectively, the index operand is always taken modulo VLEN before being used as the index.

It is not illegal to perform insert operations when VLEN(VFP_WIDTH, P) = 1.

Pseudo Code

```
                                        /* VFDINS*/
```

Assembly Code Example

```
VFDINS fd[u5], fs1          ; insert scalar into a vector at literal index

VFDINS fd[b], fs1           ; insert scalar into a vector at variable index
```

Syntax and Encoding

		Instruction Code
VFDINS	fd [u5], fs1	11100uuu0101000110UU0EEEEE1LLLLL
VFDINS	fd[b], fs1	11100bbb1000000110BB0EEEEE1LLLLL

VFDMADD

Function

Double-precision floating-point vector fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding.

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector + (vector x vector)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFDMADD fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s1 = -infinity or (s2 * s1)= -infinity and s1 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vector operands s2 and s1 is computed, added to s3, 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

```
fd = s3 + (s1 * s2)                          /* VFDMADD */
```

Assembly Code Example

```
VFDMADD fd, s3, s1, s2     ; vector floating-point multiplication of s2
                           and s1 and the ; product is accumulated and the
                           sum stored in s3.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDMADD	s3, s1, s2	11100mmm00000010 01MM1EEEEE0LLLLL
		11100mmm00000010 10MM1EEEEE0LLLLL

VFDMADDS

Function

Double-precision floating-point vector and scalar fusedMultiplyAdd. A vector is multiplied by a scalar, and the product is added to another vector. The sum is stored in the destination register

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
(vector + (vector x scalar))
```

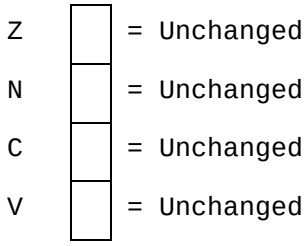
Instruction Format

op fd, s3, s1, s2

Syntax Example

VFDMADDS fd, s3, s1, s2

STATUS32 Flags Affected



FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = -infinity or (s2 * s1)= -infinity and s3 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vector s1 is multiplied by the scalar s2, and the resulting product is added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = (s3 + (s1 * s2))                    /* VFDMADDS */
```

Assembly Code Example

```
VFDMADDS fd, s3, s1, s2    ; vector floating-point multiplication of s1
                           and scalar s2 and the product is added to s3
                           and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDMADDS	s1, s2, s3	11100mmm00000110 10MM 1EEEEE 0LLLLL

VFDMOVCC

Function

Move contents from all the double-precision elements in the source vector to the destination vector if the conditions are met

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

fd = (cond(cc) == true) ? s1

Instruction Format

op fd, s1

Syntax Example

VFDMOV<.cc> fd, s1

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Move contents from the double-precision elements in the source vector to the destination vector if the conditions are met on all the elements in the vector. Either all the elements are copied or none of the elements are copied.

Pseudo Code

```
fd = (cond(cc) == true) ? s1          /* VFDMOV */
```

Assembly Code Example

```
VFDMOV fd,s1        ; Move contents of fs1 into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDMOV<.cc>	fd, s1	11100qqq01001001 10QQ0EEEEE1LLLLL

VFDMSUB

Function

Double-precision floating-point vector fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding.

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector – (vector x vector)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFDMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 product of vector operands s1 and s2 is computed, the product is subtracted from s3; the result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)     /* VFDMSUB */
```

Assembly Code Example

```
VFDMSUB fd, s3, s1, s2    ; The product of vector operands s1 and s2 is
                          computed, the product is subtracted from s3; the
                          result is rounded and stored in the destination
                          register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDMSUB	fd, s1, s2, s3	11100mmm00000010 10MM 1EEEEE 1LLLLL

VFDMSUBS

Function

Double-precision floating-point vector and a scalar fusedMultiplySubtract. A fusedMultiplySubtract is a vector floating-point and scalar multiply–subtract operation performed in one step, with a single rounding.

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector – (vector x scalar)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFDMSUBS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

A vector is multiplied by a scalar, and the product is subtracted from another vector. The difference is stored in the destination register

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)     /* VFDMSUBS */
```

Assembly Code Example

```
VFDMSUBS fd, s3, s1, s2   ; A vector is multiplied by a scalar, and the
                          product is subtracted from another vector. The
                          difference is stored in the destination register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDMSUBS	fd, s1, s2, s3	11100mmm00000110 10MM 1EEEEE 1LLLLL

VFDMUL

Function

Double-precision vector floating-point multiplication

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 * s2
```

Instruction Format

op fd,s1,s2

Syntax Example

VFDMUL fd,s1,s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 double-precision elements in a vector are multiplied with the corresponding double-precision elements in the second vector. The result is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* VFDMUL */
```

Assembly Code Example

```
VFDMUL fd,s1,s2      ; floating-point multiplication of s1 and s2 and the
                     ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDMUL	fd, s1, s2	11100mmm00010010 10MM0EEEEE1LLLLL

VFDMULS

Function

Multiply each double-precision element vector floating-point number with a scalar

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 * s2
```

Instruction Format

op fd,s1,s2

Syntax Example

VFDMULS fd,s1,s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Each double-precision element in a vector floating-point number is multiplied by a scalar. The result is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* VFDMULS */
```

Assembly Code Example

```
VFDMULS fd,s1,s2     ; floating-point multiplication of vector s1 and scalar s2
                     and the ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDMULS	fd, s1, s2	11100mmm00010110 10MM0EEEEE1LLLLL

VFDNMADD

Function

Double-precision floating-point vector fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding. The negation of the result is stored in the destination register

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
– (vector + (vector x vector))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFDNMADD fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and a = -infinity or (s2 * s3)= -infinity and a = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vectors s1 and s2 is computed, added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 + (s1 * s2))                    /* VFDNMADD */
```

Assembly Code Example

```
VFDNMADD fd, s3, s1, s2    ; vector floating-point multiplication of s2
                           and s3 and the product is added to s1 and the
                           negated result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDNMADD	s1, s2, s3	11100mmm00000111 0MM1EEEEE0LLLLL

VFDNMADDS

Function

Double-precision floating-point vector and scalar fusedMultiplyAdd. A vector is multiplied by a scalar, and the product is added to another vector. The negation of the sum is stored in the destination register

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
– (vector + (vector x scalar))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFDNMADDS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = -infinity or (s2 * s1)= -infinity and s3 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vector s1 is multiplied by the scalar s2, and the resulting product is added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 + (s1 * s2))                    /* VFDNMADDS */
```

Assembly Code Example

```
VFDNMADDS fd, s3, s1, s2   ; vector floating-point multiplication of s1
                           and scalar s2 and the product is added to s3
                           and the negated result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDNMADDS	s1, s2, s3	11100mmm00000111 10MM 1EEEEE 0LLLLL

VFDNMSUB

Function

Double-precision floating-point vector fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding. The negated result is stored in the destination register

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
-(vector – (vector x vector))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFDNMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 product of vector operands s1 and s2 is computed, the product is subtracted from s3; the negated result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))       /* VFDNMSUB */
```

Assembly Code Example

```
VFDNMSUB fd, s3, s1, s2   ; The product of vector operands s1 and s2 is
                          computed, the product is subtracted from s3; the
                          negated result is rounded and stored in the
                          destination register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDNMSUB	fd, s1, s2, s3	11100mmm00000011 10MM 1EEEEE 1LLLLL

VFDNMSUBS

Function

Double-precision floating-point vector and a scalar fusedMultiplySubtract. A fusedMultiplySubtract is a vector floating-point and scalar multiply–subtract operation performed in one step, with a single rounding. The negated result is stored in the destination register

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
-(vector – (vector x scalar))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFDNMSUBS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

A vector is multiplied by a scalar, and the product is subtracted from another vector. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))     /* VFDNMSUBS */
```

Assembly Code Example

```
VFDNMSUBS fd, s3, s1, s2    ; A vector is multiplied by a scalar, and the
                            product is subtracted from another vector. The
                            negated result is stored in the destination
                            register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDNMSUBS	fd, s1, s2, s3	11100mmm00000111 10MM 1EEEEE 1LLLLL

VFDREP

Function

Replicate a double-precision value across all double-precision elements of a vector

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION ==TRUE)

Operation

Instruction Format

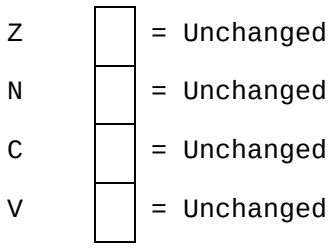
op fd, fs1

Syntax Example

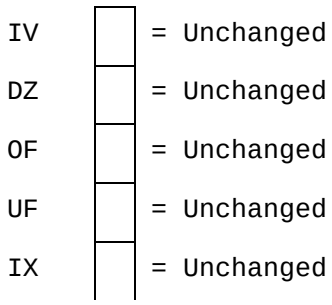
VFDREP fd, fs1

STATUS32 Flags Affected

STATUS32 Flags



FPU_STATUS Flags



Description

This instruction replicates a double-precision value across all double-precision elements of a vector.

The number of consecutive vector elements contained in a floating-point register is a function VLEN, which depends on VFP_WIDTH and the size of the floating-point type given by the P operand. That is, VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P).

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, each source or destination vector operand for these vector insert/extract/replicate operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

It is not illegal to perform replicate operations when VLEN(VFP_WIDTH, P) = 1.

Pseudo Code

```
                                        /* VFDREP*/
```

Assembly Code Example

```
VFDREP fd, fs1              ; replicate double-precision value across all
                            double-precision elements of a vector
```

Syntax and Encoding

		Instruction Code
VFDREP	fd, fs1	11100000010100101000000EEEEE1LLLLL

VFDSQRT

Function

Double-precision floating-point square-root of all the elements in a vector

Extension Group

(FP_DP_OPTION == TRUE) && (FPU_DIV_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = sqrt (s1)
```

Instruction Format

op fd, s1

Syntax Example

VFDSQRT fd, fs1

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 fd operand contains the square-root of all the half-precision elements of the s1 operand.

Pseudo Code

```
fd = sqrt (s1)                          VFDSQRT
```

Assembly Code Example

```
VFDSQRT fd, s1      fd operand contains the square-root of the fs1 operand
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFDSQRT           fd, s1       11100000010000011000 0EEEEE 1LLLLL
```

VFDSUB

Function

Vector double-precision floating-point subtraction

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1-s2
```

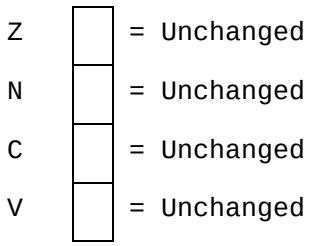
Instruction Format

op fd, s1, s2

Syntax Example

VFDSUB fd, s1, s2

STATUS32 Flags Affected



FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 double-precision elements in a vector are subtracted from the corresponding half-precision elements in the second vector. The difference is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*VFHSUB
```

Assembly Code Example

```
VFDSUB fd, s1, s2    difference of s1 and s2 operands is stored in fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFDSUB            fd, s1, s2    11100mmm00010001 10MM0EEEEE1LLLLL
```

VFDSUBS

Function

Subtract a scalar from a vector with double-precision floating-point elements

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1-s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFDSUBS fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 scalar value is subtracted from each double-precision element in a vector. The difference is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*VFDSUBS
```

Assembly Code Example

```
VFDSUBS fd, s1, s2    ; vector floating-point subtraction of s1 and scalar s2
                      and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFDSUBS            fd, s1, s2    11100mmm00010101000MM0EEEEE1LLLLL
```

VFDSUBADD

Function

Instruction performs double-precision subtraction in all even-numbered lanes and addition in all odd-numbered lanes.

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

On even-numbered lanes:

```
fd = s1 - s2
```

On odd-numbered elements:

```
fd = s1 + s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFDSUBADD fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Instruction performs subtraction in all even-numbered lanes and addition in all odd-numbered lanes.

Pseudo Code

```
even-numbered lanes:                          /* VFDSUBADD */
  fd = s1 - s2
odd-numbered lanes:
  fd = s1 + s2
```

Assembly Code Example

```
VFDSUBADD fd, s1, s2    ;Instruction performs subtraction in all even-
                        numbered lanes and addition in all odd-numbered
                        lanes.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFDSUBADD	fd, s1, s2	11100mmm00011111110MM0EEEEE1LLLLL

VFHADD

Function

Half-precision floating-point vector addition

Extension Group

(FP_HP_OPTION == TRUE) &&(FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 +s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFHADD fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The half-precision elements in a vector are added with the corresponding half-precision elements in the second vector. The sum is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* VFHADD */
```

Assembly Code Example

```
VFHADD fd, s1, s2     ; vector floating-point addition of s1 and s2
                      and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHADD	fd, s1, s2	11100mmm00010000000MM0EEEEE1LLLLL

VFHADDS

Function

Add a scalar to a vector with half-precision floating-point elements

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 +s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFHADD fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The scalar value is added with each half-precision element in a vector. The sum is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* VFHADD */
```

Assembly Code Example

```
VFHADDS fd, s1, s2    ; vector floating-point addition of s1 and
                      scalar s2 and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHADDS	fd, s1, s2	11100mmm00010000000MM0EEEEE1LLLLL

VFHADDSUB

Function

Instruction performs half-precision addition in all even-numbered lanes and subtraction in all odd-numbered lanes.

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

On even-numbered lanes:

```
fd = s1 + s2
```

On odd-numbered elements:

```
fd = s1 - s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFHADDSUB fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Instruction performs addition in all even-numbered lanes and subtraction in all odd-numbered lanes.

Pseudo Code

```
even-numbered lanes:                          /* VFHADDSUB */
  fd = s1 +s2
odd-numbered lanes:
  fd = s1 - s2
```

Assembly Code Example

```
VFHADDSUB fd, s1, s2    ;Instruction performs addition in all even-
                        numbered lanes and subtraction in all odd-
                        numbered lanes.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHADDSUB	fd, s1, s2	11100mmm00001110 00MM0EEEEE1LLLLL

VFHDIV

Function

Half-precision vector floating-point division

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFHDIV fd, s1, s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ	•	Set when the divisor is zero
OF	•	Set when the normalized result exceeds the maximumrepresentable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The half-precision elements in a vector are divided by the half-precision elements in the second vector operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                                /* VFHDIV */
```

Assembly Code Example

```
VFHDIV fd, s1, s2     ; floating-point division of s1 by s2 and the
                      ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHDIV	fd, s1, s2	11100mmm00010011100MM0EEEEE1LLLLL

VFHDIVS

Function

Divide each half-precision floating-point vector element by a scalar

Extension Group

(FP_HP_OPTION == TRUE) && (FPU_DIV_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFHDIVS fd, s1, s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ	•	Set when the divisor is zero
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

Each half-precision element in a vector is divided by a scalar operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                                /* VFHDIVS */
```

Assembly Code Example

```
VFHDIVS fd, s1, s2    ; floating-point division of a vector s1 by a
                      scalar s2 and the
                      ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHDIVS	fd, s1, s2	11100mmm00010111100MM0EEEEE1LLLLL

VFHEXT

Function

Extract half-precision element from vector at a literal index or variable index

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

Instruction Format

op fd, fs1 [u5] //extract from literal index

op fd, fs1 [b]//extract from variable index

Syntax Example

VFHEXT fd, fs1 [u5]

VFHEXT fd, fs1 [b]

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction extracts half-precision element from vector at a literal index or variable index.

The number of consecutive vector elements contained in a floating-point register is a function `VLEN`, which depends on `VFP_WIDTH` and the size of the floating-point type given by the P operand. That is, `VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P)`.

Within a floating-point vector the elements are numbered from 0 upwards.

If `VFP_WIDTH > FPR_WIDTH`, each source or destination vector operand for these vector insert/extract/replicate operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

The VF<P>EXT instructions silently ignore any bits of the u5 or B-register index operand that are not needed to index into the vector operand. Effectively, the index operand is always taken modulo `VLEN` before being used as the index.

It is not illegal to perform extract operations when `VLEN(VFP_WIDTH, P) = 1`.

Pseudo Code

```
                                        /* VFHEXT*/
```

Assembly Code Example

```
VFHEXT fd, fs1 [u5]        ; extract scalar from a vector at literal index

VFHEXT fd, fs1 [b]         ; extract scalar from a vector at variable index
```

Syntax and Encoding

		Instruction Code
VFHEXT	fd, fs1 [u5]	11100uuu010100000UU0EEEEE1LLLLL
VFHEXT	fd, fs1 [b]	11100bbb100000000BB0EEEEE1LLLLL

VFHINS

Function

Insert half-precision element into vector at a literal index or variable index

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

Instruction Format

op fd[u5], fs1 //extract from literal index

op fd [b], fs1//extract from variable index

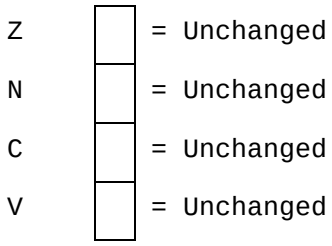
Syntax Example

VFHINS fd[u5], fs1

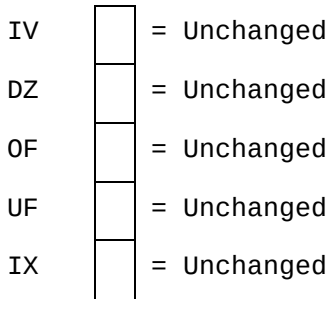
VFHINS fd [b], fs1

STATUS32 Flags Affected

STATUS32 Flags



FPU_STATUS Flags



Description

This instruction inserts half-precision element into vector at a literal index or variable index.

The number of consecutive vector elements contained in a floating-point register is a function VLEN, which depends on VFP_WIDTH and the size of the floating-point type given by the P operand. That is, VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P).

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, each source or destination vector operand for these vector insert operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

The VF<P>INS instructions silently ignore any bits of the u5 or B-register index operand that are not needed to index into the vector operand. Effectively, the index operand is always taken modulo VLEN before being used as the index.

It is not illegal to perform insert operations when VLEN(VFP_WIDTH, P) = 1.

Pseudo Code

```
                                        /* VFHINS*/
```

Assembly Code Example

```
VFHINS fd[u5], fs1          ; insert scalar into a vector at literal index

VFHINSfd[b], fs1            ; insert scalar into a vector at variable index
```

Syntax and Encoding

		Instruction Code
VFHINS	fd [u5], fs1	11100uuu0101000100UU0EEEEE1LLLLL
VFHINS	fd[b], fs1	11100bbb1000000100BB0EEEEE1LLLLL

VFHMADD

Function

Half-precision floating-point vector fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding.

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector + (vector x vector)
```

Instruction Format

op fd, s1, s2, s3

Syntax Example

VFHMADD fd, s1, s2, s3

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s3)= +infinity and s1 = -infinity or (s2 * s3)= -infinity and s1 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded after the accumulation

Description

The product of vector operands s2 and s3 is computed, added to s1, 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

```
fd = s3 + (s1 * s2)                          /* VFHMADD */
```

Assembly Code Example

```
VFHMADD fd, s1, s2, s3     ; vector floating-point multiplication of s2
                           and s3 and the ; product is accumulated and the
                           sum stored in s1.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHMADD	s1, s2, s3	11100mmm00000010 00MM1EEEEE0LLLLL

VFHMADDS

Function

Half-precision floating-point vector and scalar fusedMultiplyAdd. A vector is multiplied by a scalar, and the product is added to another vector. The sum is stored in the destination register

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
(vector + (vector x scalar))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFHMADDS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s1)= +infinity and s3 = -infinity or (s2 * s1)= -infinity and s3 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded after the accumulation

Description

The vector s1 is multiplied by the scalar s2, and the resulting product is added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = (s3 + (s1 * s2))                    /* VFHMADDS */
```

Assembly Code Example

```
VFHMADDS fd, s3, s1, s2    ; vector floating-point multiplication of s1
                           and scalar s2 and the product is added to s3
                           and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHMADDS	s1, s2, s3	11100mmm00000011000MM1EEEEE0LLLLL

VFHMOVCC

Function

Move contents from all the half-precision elements in the source vector to the destination vector if the conditions are met

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

fd = (cond(cc) == true) ? s1

Instruction Format

op fd, s1

Syntax Example

VFHMOV<.cc> fd, s1

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Move contents from the half-precision elements in the source vector to the destination vector if the conditions are met on all the elements in the vector. Either all the elements are copied or none of the elements are copied.

Pseudo Code

```
fd = (cond(cc) == true) ? s1          /* VFHMOV */
```

Assembly Code Example

```
VFHMOV fd,s1        ; Move contents of fs1 into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHMOV<.cc>	fd, s1	11100qqq01001001 00QQ0EEEEE1LLLLL

VFHMSUB

Function

Half-precision floating-point vector fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding.

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector – (vector x vector)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFHMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The product of vector operands s1 and s2 is computed, the product is subtracted from s3; the result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)     /* VFHMSUB */
```

Assembly Code Example

```
VFHMSUB fd, s3, s1, s2    ; The product of vector operands s1 and s2 is
                          computed, the product is subtracted from s3; the
                          result is rounded and stored in the destination
                          register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHMSUB	fd, s1, s2, s3	11100mmm00000010 00MM1EEEEE1LLLLL

VFHMSUBS

Function

Half-precision floating-point vector and a scalar fusedMultiplySubtract. A fusedMultiplySubtract is a vector floating-point and scalar multiply–subtract operation performed in one step, with a single rounding.

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector – (vector x scalar)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFHMSUBS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

A half-precision vector is multiplied by a scalar, and the product is subtracted from another vector. The difference is stored in the destination register

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)     /* VFHMSUBS */
```

Assembly Code Example

```
VFHMSUBS fd, s3, s1, s2   ; A vector is multiplied by a scalar, and the
                          product is subtracted from another vector. The
                          difference is stored in the destination register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHMSUBS	fd, s1, s2, s3	11100mmm00000110000MM1EEEEE1LLLLL

VFHMUL

Function

Half-precision vector floating-point multiplication

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 * s2
```

Instruction Format

op fd,s1,s2

Syntax Example

VFHMUL fd,s1,s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		= Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The half-precision elements in a vector are multiplied with the corresponding half-precision elements in the second vector. The result is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* VFHMUL */
```

Assembly Code Example

```
VFHMUL fd,s1,s2      ; floating-point multiplication of s1 and s2 and the
                     ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHMUL	fd, s1, s2	11100mmm00010010 00MM0EEEEE1LLLLL

VFHMULS

Function

Multiply each half-precision element vector floating-point number with a scalar

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 * s2
```

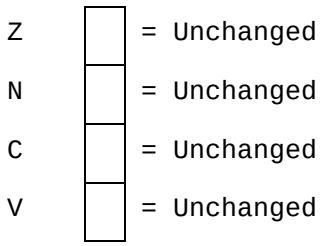
Instruction Format

op fd,s1,s2

Syntax Example

VFHMULS fd,s1,s2

STATUS32 Flags Affected



FPU_STATUS Flags

IV	•	Set when (0*infinity) or (infinity*0) operations occur Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		= Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

Each half-precision element in a vector floating-point number is multiplied by a scalar. The result is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* VFHMULS */
```

Assembly Code Example

```
VFHMULS fd,s1,s2     ; floating-point multiplication of vector s1 and scalar s2
                     and the ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHMULS	fd, s1, s2	11100mmm00010110 00MM0EEEEE1LLLLL

VFHNMADD

Function

Half-precision floating-point vector fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding. The negation of the result is stored in the destination register

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
– (vector + (vector x vector))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFHNMADD fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and a = -infinity or (s2 * s3)= -infinity and a = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded after the accumulation

Description

The product of vectors s1 and s2 is computed, added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 + (s1 * s2))                    /* VFHNMADD */
```

Assembly Code Example

```
VFHNMADD fd, s3, s1, s2    ; vector floating-point multiplication of s1
                           and s2, the product is added to s1 and the
                           negated result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHNMADD	s1, s2, s3	11100mmm00000011 00MM1EEEEE0LLLLL

VFHNMADDS

Function

Half-precision floating-point vector and scalar fusedMultiplyAdd. A vector is multiplied by a scalar, and the product is added to another vector. The negation of the sum is stored in the destination register

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
– (vector + (vector x scalar))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFHNMADDS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and a = -infinity or (s2 * s3)= -infinity and a = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded after the accumulation

Description

The vector s1 is multiplied by the scalar s2, and the resulting product is added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 + (s1 * s2))                    /* VFHNMADDS */
```

Assembly Code Example

```
VFHNMADDS fd, s3, s1, s2   ; vector floating-point multiplication of s1
                           and scalar s2 and the product is added to s3
                           and the negated result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHNMADDS	s1, s2, s3	11100mmm00000111100MM1EEEEE0LLLLL

VFHNMSUB

Function

Half-precision floating-point vector fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding. The negated result is stored in the destination register

Extension Group

(FP_HP_OPTION == TRUE && (FP_VEC_OPTION==TRUE)

Operation

```
-(vector – (vector x vector))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFHNMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The product of vector operands s1 and s2 is computed, the product is subtracted from s3; the negated result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))        /* VFHNMSUB */
```

Assembly Code Example

```
VFHNMSUB fd, s3, s1, s2   ; The product of vector operands s1 and s2 is
                          computed, the product is subtracted from s3; the
                          negated result is rounded and stored in the
                          destination register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHNMSUB	fd, s1, s2, s3	11100mmm00000011 00MM1EEEEE1LLLLL

VFHNMSUBS

Function

Half-precision floating-point vector and a scalar fusedMultiplySubtract. A fusedMultiplySubtract is a vector floating-point and scalar multiply–subtract operation performed in one step, with a single rounding. The negated result is stored in the destination register

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
-(vector – (vector x scalar))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFHNMSUBS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

A vector is multiplied by a scalar, and the product is subtracted from another vector. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))            /* VFHNMSUBS */
```

Assembly Code Example

```
VFHNMSUBS fd, s3, s1, s2    ; A vector is multiplied by a scalar, and the
                            product is subtracted from another vector. The
                            negated result is stored in the destination
                            register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHNMSUBS	fd, s1, s2, s3	11100mmm00000111 00MM1EEEEE1LLLLL

VFHEXCH

Function

Vector exchange permutation operation on the half-precision elements in the source register to the destination register

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = { sn-1, sn-2, …, s1, toggle(s0)}
```

Note

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is a copy of *S* with bit s0 toggled.

Instruction Format

op fd, b

Syntax Example

VFHEXCH fd, fs0

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction performs the exchange permutation operation on the source operand. The destination register index is a copy of the source destination index with bit s0 (source index 0) toggled.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-2 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFHEXCH	b6	b7	b4	b5	b2	b3	b0	b1		b2	b3	b0	b1		b0	b1

Pseudo Code

```
D = { sn-1, sn-2, …, s1, toggle(s0)}          /* VFHEXCH*/
```

Assembly Code Example

```
VFHEXCH fd, fs0          ;exchange operations on the elements of the
                         source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFHEXCH	fd, fs0, fs1	11100mmm000111000MM0EEEEE1LLLLL

VFDEXCH

Function

Vector exchange permutation operation on the double-precision elements in the source register to the destination register

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = { sn-1, sn-2, …, s1, toggle(s0)}
```

Note *S* is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is a copy of *S* with bit s0 toggled.

Instruction Format

op fd, b

Syntax Example

VFDEXCH fd, fs0

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction performs the exchange permutation operation on the source operand. The destination register index is a copy of the source destination index with bit s0 (source index 0) toggled.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-3 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFDEXCH	b3	b2	b1	b0	b7	b6	b5	b4		Illegal					Illegal	

Pseudo Code

```
D = { sn-1, sn-2, …, s1, toggle(s0)}          /* VFDEXCH*/
```

Assembly Code Example

```
VFDEXCH fd, fs0          ;exchange operations on the elements of the
                         source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFDEXCH	fd, fs0	11100000010000101000EEEEE1LLLLL

VFSEXCH

Function

Vector exchange permutation operation on the single-precision elements in the source register to the destination register

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = { sn-1, sn-2, …, s1, toggle(s0)}
```

Note

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is a copy of *S* with bit s0 toggled.

Instruction Format

op fd, b

Syntax Example

VFSEXCH fd, fs0

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction performs the exchange permutation operation on the source operand. The destination register index is a copy of the source destination index with bit s0 (source index 0) toggled.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-4 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFSEXCH	b5	b4	b7	b6	b1	b0	b3	b2		b1	b0	b3	b2		Illegal	

Pseudo Code

```
D = { sn-1, sn-2, …, s1, toggle(s0)}          /* VFSEXCH*/
```

Assembly Code Example

```
VFSEXCH fd, fs0          ;exchange operations on the elements of the
                         source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFSEXCH	fd, fs0	11100000010000101000EEEEE1LLLLL

VFHBFLYL

Function

Vector butterfly operation on the half-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = {s0, sn-2, …, s1, sn-1}
```

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is a copy of *S* with bits *s0* and *sn-1* interchange.

Instruction Format

op fd, b, c

Syntax Example

VFHBFLYL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction performs butterfly operation on the source operands. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-5 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFHBFLYL	c6	b6	c4	b4	c2	b2	c0	b0		c2	b2	c0	b0		c0	b0

Pseudo Code

```
D = {s0, sn-2, …, s1, sn- }                    /* VFHBFLYL*/
```

Assembly Code Example

```
VFHBFLYL fd, fs0, fs1      ;butterfly operations on the elements of the
                           source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFHBFLYL	fd, fs0, fs1	11100mmm000111000MM0EEEEE1LLLLL

VFSBFLYL

Function

Vector butterfly operation on the single-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = {s0, sn-2, …, s1, sn-1}
```

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is a copy of *S* with bits *s0* and *sn-1* interchange.

Instruction Format

op fd, b, c

Syntax Example

VFSBFLYL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction performs butterfly operation on the source operands. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-6 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFSBFLYL	c6	b6	c4	b4	c2	b2	c0	b0		c2	b2	c0	b0		c0	b0

Pseudo Code

```
D = {s0, sn-2, …, s1, sn- }                    /* VFHBFLYL*/
```

Assembly Code Example

```
VFSBFLYL fd, fs0, fs1      ;butterfly operations on the elements of the
                           source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFSBFLYL	fd, fs0, fs1	11100mmm000111100 01MM0EEEEE1LLLLL

VFDBFLYL

Function

Vector butterfly operation on the double-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = {s0, sn-2, …, s1, sn-1}
```

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is a copy of *S* with bits *s0* and *sn-1* interchange.

Instruction Format

op fd, b, c

Syntax Example

VFDBFLYL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX [ ] = Unchanged

Description

This instruction performs butterfly operation on the source operands. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-7 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFDBFLYL	c3	c2	c1	c0	b3	b2	b1	b0		Illegal					Illegal	

Pseudo Code

```
D = {s0, sn-2, …, s1, sn- }                    /* VFHBFLYL*/
```

Assembly Code Example

```
VFDBFLYL fd, fs0, fs1      ;butterfly operations on the elements of the
                           source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFDBFLYL	fd, fs0, fs1	11100mmm000111100 10MM0EEEEE1LLLLL

VFHBFLYM

Function

Vector butterfly operation on the half-precision elements in the source register to the destination register, and store the most-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = {s0, sn-2, …, s1, sn-1}
```

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is a copy of *S* with bits *s0* and *sn-1* interchange.

Instruction Format

op fd, b, c

Syntax Example

VFHBFLYM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction performs butterfly operation on the source operands. The most-significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-8 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFHBFLYM	c7	b7	c5	b5	c3	b3	c1	b1		c3	b3	c1	b1		c1	b1

Pseudo Code

```
D = {s0, sn-2, …, s1, sn- }                    /* VFHBFLYM*/
```

Assembly Code Example

```
VFHBFLYM fd, fs0, fs1      ;butterfly operations on the elements of the
                           source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFHBFLYM	fd, fs0, fs1	11100mmm000111010 0MM0EEEEE1LLLLL

VFSBFLYM

Function

Vector butterfly operation on the single-precision elements in the source register to the destination register, and store the most-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = {s0, sn-2, …, s1, sn-1}
```

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is a copy of *S* with bits *s0* and *sn-1* interchange.

Instruction Format

op fd, b, c

Syntax Example

VFSBFLYM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction performs butterfly operation on the source operands. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-9 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFSBFLYM	c7	c6	b7	b6	c3	c2	b3	b2		c3	c2	b3	b2		Illegal	

Pseudo Code

```
D = {s0, sn-2, …, s1, sn- }                    /* VFHBFLYM*/
```

Assembly Code Example

```
VFSBFLYM fd, fs0, fs1      ;butterfly operations on the elements of the
                           source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFSBFLYM	fd, fs0, fs1	11100mmm00011101 01MM0EEEEE1LLLLL

VFDBFLYM

Function

Vector butterfly operation on the double-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

```
D = {s0, sn-2, …, s1, sn-1}
```

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is a copy of *S* with bits *s0* and *sn-1* interchange.

Instruction Format

op fd, b, c

Syntax Example

VFDBFLYM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction performs butterfly operation on the source operands. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-10 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFDBFLYM	c7	c6	c5	c4	b7	b6	b5	b4		Illegal					Illegal	

Pseudo Code

```
D = {s0, sn-2, …, s1, sn- }                    /* VFHBFLYM*/
```

Assembly Code Example

```
VFDBFLYM fd, fs0, fs1      ;butterfly operations on the elements of the
                           source operands into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFDBFLYM	fd, fs0, fs1	11100mmm000111101 0MM0EEEEE1LLLLL

VFHPACKL

Function

Vector pack, that is, shuffle the half-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE_

Operation

$D = \{ s_{n-2}, \ldots, s_0, s_{n-1} \}$

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFHPACKL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction shuffles the source operands to the source operand. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-11 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFHPACKL	c3	b3	c2	b2	c1	b1	c0	b0		c1	b1	c0	b0		c0	b0

Pseudo Code

```
D = {s0, sn-1, …, s1}                    /* VFHPACKL*/
```

Assembly Code Example

```
VFHPACKL fd, fs0, fs1      ;shuffle elements of the source operands into
                           the destination operand.
```

Syntax and Encoding

		Instruction Code
VFHPACKL	fd, fs0, fs1	11100mmm000110100MM0EEEEE1LLLLL

VFSPACKL

Function

Vector pack, that is, shuffle the single-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

> **Note** *S* is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFSPACKL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction shuffles the source operands to the source operand. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-12 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFSPACKL	c3	c2	b3	b2	c1	c0	b1	b0		c1	c0	b1	b0		Illegal	

Pseudo Code

```
D = {s0, sn-1, …, s1}                    /* VFSPACKL*/
```

Assembly Code Example

```
VFSPACKL fd, fs0, fs1      ;shuffle elements of the source operands into
                           the destination operand.
```

Syntax and Encoding

		Instruction Code
VFSPACKL	fd, fs0, fs1	11100mmm000110100 1MM0EEEEE1LLLLL

VFDPACKL

Function

Vector pack, that is, shuffle the double-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

> **Note** *S* is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFDPACKL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction shuffles the double-precision data in the source operands to the source operand. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-13 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFDPACKL	c3	c2	c1	c0	b3	b2	b1	b0		Illegal					Illegal	

Pseudo Code

```
D = {s0, sn-1, …, s1}                    /* VFDPACKL*/
```

Assembly Code Example

```
VFDPACKL fd, fs0, fs1      ;shuffle elements of the source operands into
                           the destination operand.
```

Syntax and Encoding

		Instruction Code
VFDPACKL	fd, fs0, fs1	11100mmm000110101 0MM0EEEEE1LLLLL

VFHPACKM

Function

Vector pack, that is, shuffle the half-precision elements in the source register to the destination register, and store the most-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

> **Note** *S* is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFHPACKM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction shuffles the source operands to the source operand. The most significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-14 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFHPACKM	c7	b7	c6	b6	c5	b5	c4	b4		c3	b3	c2	b2		c1	b1

Pseudo Code

$D = \{s_0, s_{n-1}, \ldots, s_1\}$ /* VFHPACKM*/

Assembly Code Example

```
VFHPACKM fd, fs0, fs1      ; shuffle elements of the source operands into
                           the destination operand.
```

Syntax and Encoding

		Instruction Code
VFHPACKM	fd, fs0, fs1	11100mmm00011011 00MM0EEEEE1LLLLL

VFSPACKM

Function

Vector unpack, that is, shuffle the single-precision elements in the source register to the destination register, and store the most-significant half result

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFSPACKM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction shuffles the source operands to the source operand. The most-significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-15 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFSPACKM	c7	c6	b7	b6	c5	c4	b5	b4		c3	c2	b3	b2		Illegal	

Pseudo Code

```
D = {s0, sn-1, …, s1}                    /* VFSPACKM*/
```

Assembly Code Example

```
VFSPACKM fd, fs0, fs1      ;inverse shuffle elements of the source operands
                           into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFSPACKM	fd, fs0, fs1	11100mmm00011011 01MM0EEEEE1LLLLL

VFDPACKM

Function

Vector unpack, that is, shuffle the double-precision elements in the source register to the destination register, and store the most-significant half result

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFDPACKM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction shuffles the double-precision data in the source operands to the source operand. The most-significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-16 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFDPACKM	c7	c6	c5	c4	b7	b6	b5	b4		Illegal					Illegal	

Pseudo Code

```
D = {s0, sn-1, …, s1}                    /* VFDPACKM*/
```

Assembly Code Example

```
VFDPACKM fd, fs0, fs1      ;inverse shuffle elements of the source operands
                           into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFDPACKM	fd, fs0, fs1	11100mmm000110111 0MM0EEEEE1LLLLL

VFHUNPKL

Function

Vector unpack, that is, inverse shuffle the half-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFHUNPKL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction inverse shuffles the source operands to the source operand. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-17 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFHUNPKL	c6	c4	c2	c0	b6	b4	b2	b0		c2	c0	b2	b0		c0	b0

Pseudo Code

```
D = {s0, sn-1, ..., s1}                    /* VFHUNPKL*/
```

Assembly Code Example

```
VFHUNPKL fd, fs0, fs1      ;inverse shuffle elements of the source operands
                           into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFHUNPKL	fd, fs0, fs1	11100mmm000110000 0MM0EEEEE1LLLLL

VFSUNPKL

Function

Vector unpack, that is, inverse shuffle the single-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

Note *S* is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFSUNPKL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction inverse shuffles the source operands to the source operand. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-18 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFSUNPKL	c5	c4	c1	c0	b5	b4	b1	b0		c1	c0	b1	b0		Illegal	

Pseudo Code

```
D = {s0, sn-1, ..., s1}                    /* VFSUNPKL*/
```

Assembly Code Example

```
VFSUNPKL fd, fs0, fs1      ;inverse shuffle elements of the source operands
                           into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFSUNPKL	fd, fs0, fs1	11100mmm000110000 1MM0EEEEE1LLLLL

VFDUNPKL

Function

Vector unpack, that is, inverse shuffle the double-precision elements in the source register to the destination register, and store the least-significant half result

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

> **Note** *S* is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the left circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFDUNPKL fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction inverse shuffles the double-precision data in the source operands to the source operand. The least significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-19 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFDUNPKL	c3	c2	c1	c0	b3	b2	b1	b0		Illegal					Illegal	

Pseudo Code

```
D = {s0, sn-1, …, s1}                    /* VFDUNPKL*/
```

Assembly Code Example

```
VFDUNPKL fd, fs0, fs1      ;inverse shuffle elements of the source operands
                           into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFDUNPKL	fd, fs0, fs1	11100mmm000110000MM0EEEEE1LLLLL

VFHUNPKM

Function

Vector unpack, that is, inverse shuffle the half-precision elements in the source register to the destination register, and store the most-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the right circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFHUNPKM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction inverse shuffles the source operands to the source operand. The most significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-20 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFHUNPKM	c7	c5	c3	c1	b7	b5	b3	b1		c3	c1	b3	b1		c1	b1

Pseudo Code

```
D = {s0, sn-1, ..., s1}                    /* VFHUNPKM*/
```

Assembly Code Example

```
VFHUNPKM fd, fs0, fs1      ;inverse shuffle elements of the source operands
                           into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFHUNPKM	fd, fs0, fs1	11100mmm00011001 00MM0EEEEE1LLLLL

VFSUNPKM

Function

Vector unpack, that is, inverse shuffle the single-precision elements in the source register to the destination register, and store the most-significant half result

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

Note *S* is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function. In this operation, *D* is the right circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFSUNPKM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction inverse shuffles the source operands to the source operand. The most-significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-21 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFSUNPKM	c7	c6	c3	c2	b7	b6	b3	b2		c3	c2	b3	b2		Illegal	

Pseudo Code

$D = \{s_0, s_{n-1}, \ldots, s_1\}$ /* VFSUNPKM*/

Assembly Code Example

```
VFSUNPKM fd, fs0, fs1      ;inverse shuffle elements of the source operands
                           into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFSUNPKM	fd, fs0, fs1	11100mmm00011001 01MM0EEEEE1LLLLL

VFDUNPKM

Function

Vector unpack, that is, inverse shuffle the double-precision elements in the source register to the destination register, and store the most-significant half result

Extension Group

(FP_DP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

$D = \{s_0, s_{n-1}, \ldots, s_1\}$

S is the source element index (n-bit value) and *D* (n-bit value) denotes the destination element index. For each permutation instruction, any source index *S* maps to a unique destination index *D* according to its permutation function.
In this operation, *D* is the right circular rotation of *S*.

Instruction Format

op fd, b, c

Syntax Example

VFDUNPKM fd, fs0, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged

IX = Unchanged

Description

This instruction inverse shuffles the double-precision data in the source operands to the source operand. The most-significant elements are returned in the destination register.

Note

- When the -fp_wide_option is true, each operand is a pair of FP registers, in common with all vector FP instructions.
- The semantics of all versions of each permutation instruction, for all values of <P> and all values of VFP_WIDTH, are enumerated in the following table. This table shows the result vector returned by each permutation instruction as an ordered vector of 16-bit elements obtained from one or both of the source vector operands. The values from the first source operand are b0 through b7, and from the second source operand are c0 through c7. When VFP_WIDTH is 64 or 32 there are only 4 or 2 such 16-bit elements per vector operand respectively. Instructions with <P> set for single-precision permute the source operands in pairs of 16-bit values. Similarly, with <P> set for double-precision, 16-bit elements are combined into groups of 4.

Table 39-22 Permutation Instruction Result

Permutation Instruction	Result vector lane contents when VFP_WIDTH is 128, 64, 32															
	128									64					32	
	7	6	5	4	3	2	1	0		3	2	1	0		1	0
VFDUNPKM	c7	c6	c5	c4	b7	b6	b5	b4		Illegal					Illegal	

Pseudo Code

```
D = {s0, sn-1, ..., s1}                    /* VFDUNPKM*/
```

Assembly Code Example

```
VFDUNPKM fd, fs0, fs1      ;inverse shuffle elements of the source operands
                           into the destination operand.
```

Syntax and Encoding

		Instruction Code
VFDUNPKM	fd, fs0, fs1	11100mmm00011001 10MM0EEEEE1LLLLL

VFHREP

Function

Replicate a half-precision value across all half-precision elements of a vector

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

Instruction Format

op fd, fs1

Syntax Example

VFHREP fd, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction replicates a half-precision value across all half-precision elements of a vector.

The number of consecutive vector elements contained in a floating-point register is a function VLEN, which depends on VFP_WIDTH and the size of the floating-point type given by the P operand. That is, VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P).

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, each source or destination vector operand for these vector insert/extract/replicate operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

It is not illegal to perform replicate operations when VLEN(VFP_WIDTH, P) = 1.

Pseudo Code

```
                                        /* VFHREP*/
```

Assembly Code Example

```
VFHREP fd, fs1              ; replicate half-precision value across all
                            half-precision elements of a vector
```

Syntax and Encoding

		Instruction Code
VFHREP	fd, fs1	11100000010100100000EEEEE1LLLLL

VFHSQRT

Function

Half-precision floating-point square-root of all the elements in a vector

Extension Group

(FP_HP_OPTION == TRUE) && (FPU_DIV_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = sqrt (s1)
```

Instruction Format

op fd, s1

Syntax Example

VFHSQRT fd, s1

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		= Unchanged
OF		= Unchanged
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The fd operand contains the square-root of all the half-precision elements of the s1 operand.

Pseudo Code

```
fd = sqrt (s1)                          VFHSQRT
```

Assembly Code Example

```
VFHSQRT fd, s1      fd operand contains the square-root of the s1 operand
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFHSQRT             fd, s1        11100000010000010000EEEEE1LLLLL
```

VFHSUB

Function

Vector half-precision floating-point subtraction

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1-s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFHSUB fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The half-precision elements in a vector are subtracted from the corresponding half-precision elements in the second vector. The difference is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*VFHSUB
```

Assembly Code Example

```
VFHSUB fd, s1, s2    difference of s1 and s2 operands is stored in fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFHSUB            fd, s1, s2    11100mmm00010001000MM0EEEEE1LLLLL
```

VFHSUBS

Function

Subtract a scalar from a vector with half-precision floating-point elements

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1-s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFHSUBS fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
DZ		Unchanged
OF	•	Set when the normalized result exceeds the maximum representable number in a half-precision format
UF	•	Set when the result is too small to be represented in a half-precision format
IX	•	Set when result is rounded

Description

The scalar value is subtracted from each half-precision element in a vector. The difference is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*VFHSUBS
```

Assembly Code Example

```
VFHSUBS fd, s1, s2    ; vector floating-point subtraction of s1 and scalar s2
                      and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFHSUBS            fd, s1, s2    11100mmm00010101 00MM0EEEEE1LLLLL
```

VFHSUBADD

Function

Instruction performs half-precision subtraction in all even-numbered lanes and addition in all odd-numbered lanes.

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

On even-numbered lanes:

```
fd = s1 - s2
```

On odd-numbered elements:

```
fd = s1 + s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFHSUBADD fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Instruction performs subtraction in all even-numbered lanes and addition in all odd-numbered lanes.

Pseudo Code

```
even-numbered lanes:                          /* VFHSUBADD */
  fd = s1 - s2
odd-numbered lanes:
  fd = s1 + s2
```

Assembly Code Example

```
VFHSUBADD fd, s1, s2    ;Instruction performs subtraction in all even-
                        numbered lanes and addition in all odd-numbered
                        lanes.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFHSUBADD	fd, s1, s2	11100mmm00011111100MM0EEEEE1LLLLL

VFSADD

Function

Single-precision floating-point vector addition

Extension Group

(HAS_FP == TRUE) &&(FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 +s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFSADD fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s1= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 single-precision elements in a vector are added with the corresponding single-precision elements in the second vector. The sum is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* VFSADD */
```

Assembly Code Example

```
VFSADD fd, s1, s2   ; vector floating-point addition of s1 and s2 and the
                    ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSADD	fd, s1, s2	11100mmm00010000001MM0EEEEE1LLLLL

VFSADDS

Function

Add a single-precision floating-point vector with a scalar

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 + s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFSADDS fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s1= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 scalar value is added with each single-precision element in a vector. The sum is stored in the destination register.

Pseudo Code

```
fd = s1 +s2                                  /* VFSADDS */
```

Assembly Code Example

```
VFSADDS fd, s1, s2   ; vector floating-point addition of s1 and scalar s2 and
                     the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSADDS	fd, s1, s2	11100mmm00010100001MM0EEEEE1LLLLL

VFSADDSUB

Function

Instruction performs single-precision addition in all even-numbered lanes and subtraction in all odd-numbered lanes.

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

On even-numbered lanes:

```
fd = s1 + s2
```

On odd-numbered elements:

```
fd = s1 - s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFSADDSUB fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Instruction performs addition in all even-numbered lanes and subtraction in all odd-numbered lanes.

Pseudo Code

```
even-numbered lanes:                          /* VFSADDSUB */
  fd = s1 +s2
odd-numbered lanes:
  fd = s1 - s2
```

Assembly Code Example

```
VFSADDSUB fd, s1, s2    ;Instruction performs addition in all even-
                        numbered lanes and subtraction in all odd-
                        numbered lanes.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSADDSUB	fd, s1, s2	11100mmm00001110 01MM0EEEEE1LLLLL

VFSDIV

Function

Single-precision vector floating-point division

Extension Group

(HAS_FP == TRUE) && (FPU_DIV_OPTION == TRUE) &&(FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFSDIV fd, s1, s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 single-precision elements in a vector are divided by the single-precision elements in the second vector operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                                /* VFSDIV */
```

Assembly Code Example

```
VFSDIV fd, s1, s2     ; floating-point division of s1 by s2 and the
                      ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSDIV	fd, s1, s2	11100mmm00010011101MM0EEEEE1LLLLL

VFSDIVS

Function

Divide each single-precision floating-point vector element by a scalar

Extension Group

(HAS_FP == TRUE) && (FPU_DIV_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 / s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFSDIVS fd, s1, s2

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Each single-precision element in a vector is divided by a scalar operand, and the result is stored in the destination register.

Pseudo Code

```
fd = s1 / s2                                /* VFSDIVS */
```

Assembly Code Example

```
VFSDIVS fd, s1, s2    ; floating-point division of a vector s1 by a
                      scalar s2 and the
                      ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSDIVS	fd, s1, s2	11100mmm00010111100MM0EEEEE1LLLLL

VFSEXT

Function

Extract single-precision element from vector at a literal index or variable index

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

Instruction Format

op fd, fs1 [u5] //extract from literal index

op fd, fs1 [b]//extract from variable index

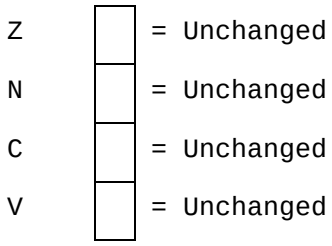
Syntax Example

VSHEXT fd, fs1 [u5]

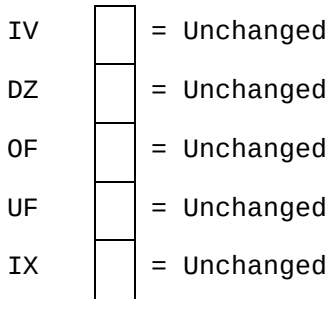
VSHEXT fd, fs1 [b]

STATUS32 Flags Affected

STATUS32 Flags



FPU_STATUS Flags



Description

This instruction extracts single-precision element from vector at a literal index or variable index.

The number of consecutive vector elements contained in a floating-point register is a function VLEN, which depends on VFP_WIDTH and the size of the floating-point type given by the P operand. That is, VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P).

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, each source or destination vector operand for these vector insert/extract/replicate operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

The VF<P>EXT instructions silently ignore any bits of the u5 or B-register index operand that are not needed to index into the vector operand. Effectively, the index operand is always taken modulo VLEN before being used as the index.

It is not illegal to perform extract operations when VLEN(VFP_WIDTH, P) = 1.

Pseudo Code

```
                                        /* VFSEXT*/
```

Assembly Code Example

```
VFSEXT fd, fs1 [u5]        ; extract scalar from a vector at literal index

VFSEXT fd, fs1 [b]         ; extract scalar from a vector at variable index
```

Syntax and Encoding

		Instruction Code
VFSEXT	fd, fs1 [u5]	11100uuu010100001UU0EEEEE1LLLLL
VFSEXT	fd, fs1 [b]	11100bbb100000001BB0EEEEE1LLLLL

VFSINS

Function

Insert single-precision element into vector at a literal index or variable index

Extension Group

(FP_HP_OPTION == TRUE) && (FP_VEC_OPTION == TRUE)

Operation

Instruction Format

op fd[u5], fs1 //extract from literal index

op fd [b], fs1//extract from variable index

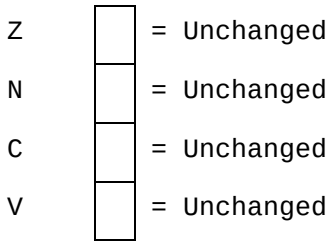
Syntax Example

VFSINS fd[u5], fs1

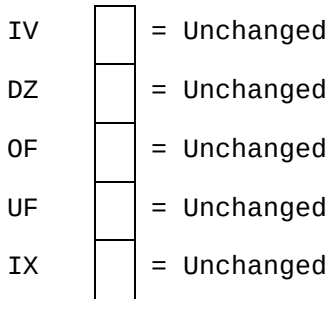
VFSINS fd [b], fs1

STATUS32 Flags Affected

STATUS32 Flags



FPU_STATUS Flags



Description

This instruction inserts single-precision element into vector at a literal index or variable index.

The number of consecutive vector elements contained in a floating-point register is a function VLEN, which depends on VFP_WIDTH and the size of the floating-point type given by the P operand. That is, VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P).

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, each source or destination vector operand for these vector insert operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

The VF<P>INS instructions silently ignore any bits of the u5 or B-register index operand that are not needed to index into the vector operand. Effectively, the index operand is always taken modulo VLEN before being used as the index.

It is not illegal to perform insert operations when VLEN(VFP_WIDTH, P) = 1.

Pseudo Code

```
                                        /* VFSINS*/
```

Assembly Code Example

```
VFSINS fd[u5], fs1          ; insert scalar into a vector at literal index

VFSINS fd[b], fs1           ; insert scalar into a vector at variable index
```

Syntax and Encoding

		Instruction Code
VFSINS	fd [u5], fs1	11100uuu0101010001 01UU0EEEEE1LLLLL
VFSINS	fd[b], fs1	11100bbb1000000101 01BB0EEEEE1LLLLL

VFSMADD

Function

Single-precision floating-point vector fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding.

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector + (vector x vector)
```

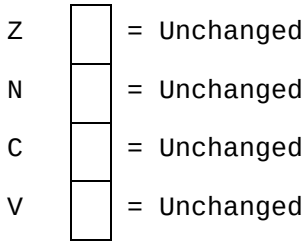
Instruction Format

op fd, s1, s2, s3

Syntax Example

VFSMADD fd, s1, s2, s3

STATUS32 Flags Affected



FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and s1 = -infinity or (s2 * s3)= -infinity and s1 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vector operands s2 and s3 is computed, added to s1, 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

```
fd = s3 + (s1 * s2)                          /* VFSMADD */
```

Assembly Code Example

```
VFSMADD fd, s1, s2, s3     ; vector floating-point multiplication of s2
                           and s3 and the ; product is accumulated and the
                           sum stored in s1.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSMADD	s1, s2, s3	11100mmm00000100 1MM1EEEEE0LLLLL

VFSMADDS

Function

Single-precision floating-point vector and scalar fusedMultiplyAdd. A vector is multiplied by a scalar, and the product is added to another vector. The sum is stored in the destination register

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
(vector + (vector x scalar))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFSMADDS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s2 operands are in the following combinations: (s2 * s1)= +infinity and s3 = -infinity or (s2 * s1)= -infinity and s3 = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vector s1 is multiplied by the scalar s2, and the resulting product is added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = (s3 + (s1 * s2))                    /* VFSMADDS */
```

Assembly Code Example

```
VFSMADDS fd, s3, s1, s2    ; vector floating-point multiplication of s1
                           and scalar s2 and the product is added to s3
                           and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSMADDS	s3, s1, s2	11100mmm00000110 01MM1EEEEE0LLLLL

VFSMOVCC

Function

Move contents from all the single-precision elements in the source vector to the destination vector if the conditions are met

Extension Group

(FP_HP_OPTION == TRUE) &&(FP_VEC_OPTION==TRUE)

Operation

fd = (cond(cc) == true) ? s1

Instruction Format

op fd, s1

Syntax Example

VFSMOV<.cc> fd, s1

STATUS32 Flags Affected

Z	•	= Set if result is zero
N	•	= Set if most-significant bit of result is set
C		= Unchanged
V		= Unchanged

Description

Move contents from the single-precision elements in the source vector to the destination vector if the conditions are met on all the elements in the vector. Either all the elements are copied or none of the elements are copied.

Pseudo Code

```
fd = (cond(cc) == true) ? s1          /* VFSMOV */
```

Assembly Code Example

```
VFSMOV fd,s1          ; Move contents of fs1 into fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSMOV<.cc>	fd, s1	11100qqq01001001 00QQ0EEEEE1LLLLL

VFSMSUB

Function

Single-precision floating-point vector fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding.

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector – (vector x vector)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFSMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vector operands s1 and s2 is computed, the product is subtracted from s3; the result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)     /* VFSMSUB */
```

Assembly Code Example

```
VFSMSUB fd, s3, s1, s2    ; The product of vector operands s1 and s2 is
                          computed, the product is subtracted from s3; the
                          result is rounded and stored in the destination
                          register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSMSUB	fd, s1, s2, s3	11100mmm00000100 1MM1EEEEE1LLLLL

VFSMSUBS

Function

Single-precision floating-point vector and a scalar fusedMultiplySubtract. A fusedMultiplySubtract is a vector floating-point and scalar multiply–subtract operation performed in one step, with a single rounding.

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
vector – (vector x scalar)
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFSMSUBS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

A single-precision vector is multiplied by a scalar, and the product is subtracted from another vector. The difference is stored in the destination register

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = s3 – (s1 * s2)     /* VFSMSUBS */
```

Assembly Code Example

```
VFSMSUBS fd, s3, s1, s2   ; A vector is multiplied by a scalar, and the
                          product is subtracted from another vector. The
                          difference is stored in the destination register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSMSUBS	fd, s1, s2, s3	11100mmm00000110001MM1EEEEE1LLLLL

VFSMUL

Function

Single-precision vector floating-point multiplication

Extension Group

(HAS_FP == TRUE) &&(FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 * s2
```

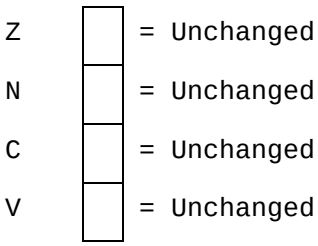
Instruction Format

op fd,s1,s2

Syntax Example

VFSMUL fd,s1,s2

STATUS32 Flags Affected



FPU_STATUS Flags

IV	•	Set when (0*infinity) or (infinity*0) operations occur Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 single-precision elements in a vector are multiplied with the corresponding single-precision elements in the second vector. The result is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* VFSMUL */
```

Assembly Code Example

```
VFSMUL fd,s1,s2      ; floating-point multiplication of s1 and s2 and the
                     ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSMUL	fd, s1, s2	11100mmm00010010 01MM0EEEEE1LLLLL

VFSMULS

Function

Multiply each single-precision element vector floating-point number with a scalar

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1 * s2
```

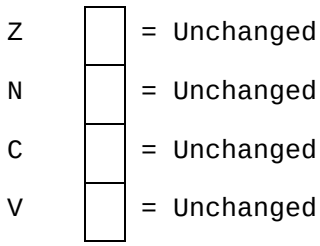
Instruction Format

op fd,s1,s2

Syntax Example

VFSMULS fd,s1,s2

STATUS32 Flags Affected



FPU_STATUS Flags

IV	•	Set when (0*infinity) or (infinity*0) operations occur Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Each single-precision element in a vector floating-point number is multiplied by a scalar. The result is stored in the destination register.

Pseudo Code

```
fd = s1 * s2                                /* VFSMULS */
```

Assembly Code Example

```
VFSMULS fd,s1,s2     ; floating-point multiplication of vector s1 and scalar s2
                     and the ; result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSMULS	fd, s1, s2	11100mmm00010110 01MM0EEEEE1LLLLL

VFSNMADD

Function

Single-precision floating-point vector fusedMultiplyAdd. A fusedMultiplyAdd is a floating-point multiply–add operation performed in one step, with a single rounding. The negation of the result is stored in the destination register

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
– (vector + (vector x vector))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFSNMADD fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and a = -infinity or (s2 * s3)= -infinity and a = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vectors s1 and s2 is computed, added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 + (s1 * s2))                    /* VFHSMADD */
```

Assembly Code Example

```
VFSNMADD fd, s3, s1, s2    ; vector floating-point multiplication of s2
                           and s3 and the product is added to s1 and the
                           negated result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSNMADD	s1, s2, s3	11100mmm00000011 01MM 1EEEEE 0LLLLL

VFSNMADDS

Function

Single-precision floating-point vector and scalar fusedMultiplyAdd. A vector is multiplied by a scalar, and the product is added to another vector. The negation of the sum is stored in the destination register

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
– (vector + (vector x scalar))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFSNMADDS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s3 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the a, b, and c operands are in the following combinations: (s2 * s3)= +infinity and a = -infinity or (s2 * s3)= -infinity and a = +infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vector s1 is multiplied by the scalar s2, and the resulting product is added to vector s3. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 + (s1 * s2))                    /* VFSNMADDS */
```

Assembly Code Example

```
VFSNMADDS fd, s3, s1, s2   ; vector floating-point multiplication of s2
                           and s3 and the product is added to s1 and the
                           negated result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSNMADDS	s1, s2, s3	11100mmm00000111101MM1EEEEE0LLLLL

VFSNMSUB

Function

Single-precision floating-point vector fusedMultiplySubtract. A fusedMultiplySubtract is a floating-point multiply–subtract operation performed in one step, with a single rounding. The negated result is stored in the destination register

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
-(vector – (vector x vector))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFSNMSUB fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 vector operands s1 and s2 is computed, the product is subtracted from s3; the negated result is rounded and stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))     /* VFSNMSUB */
```

Assembly Code Example

```
VFSNMSUB fd, s3, s1, s2   ; The product of vector operands s1 and s2 is
                          computed, the product is subtracted from s3; the
                          negated result is rounded and stored in the
                          destination register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSNMSUB	fd, s1, s2, s3	11100mmm00000011 01MM 1EEEEE 1LLLLL

VFSNMSUBS

Function

Single-precision floating-point vector and a scalar fusedMultiplySubtract. A fusedMultiplySubtract is a vector floating-point and scalar multiply–subtract operation performed in one step, with a single rounding. The negated result is stored in the destination register

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
-(vector – (vector x scalar))
```

Instruction Format

op fd, s3, s1, s2

Syntax Example

VFSNMSUBS fd, s3, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	■ Set when the s2 and s1 operands are in the following combinations: 0 * infinity or infinity * 0 ■ Set when the s1, s2, and s3 operands are in the following combinations: (s2 * s1)= +infinity and s3 = +infinity or (s2 * s1)= -infinity and s3 = -infinity ■ If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

A vector is multiplied by a scalar, and the product is subtracted from another vector. The negation of the result is stored in the destination register.

Note This instruction uses the accumulator as described in "Extension Core Registers" .

Pseudo Code

```
fd = -(s3 – (s1 * s2))      /* VFSNMSUBS */
```

Assembly Code Example

```
VFSNMSUBS fd, s3, s1, s2       ; A vector is multiplied by a scalar, and the
                               product is subtracted from another vector. The
                               negated result is stored in the destination
                               register
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSNMSUBS	fd, s1, s2, s3	11100mmm00000111 01MM 1EEEEE 1LLLLL

VFSREP

Function

Replicate a single-precision value across all single-precision elements of a vector

Extension Group

(HAS_FP ==TRUE) && (FP_VEC_OPTION == TRUE)

Operation

Instruction Format

op fd, fs1

Syntax Example

VFSREP fd, fs1

STATUS32 Flags Affected

STATUS32 Flags

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV		= Unchanged
DZ		= Unchanged
OF		= Unchanged
UF		= Unchanged
IX		= Unchanged

Description

This instruction replicates a single-precision value across all single-precision elements of a vector.

The number of consecutive vector elements contained in a floating-point register is a function VLEN, which depends on VFP_WIDTH and the size of the floating-point type given by the P operand. That is, VLEN(VFP_WIDTH, P) = VFP_WIDTH >> (4+P).

Within a floating-point vector the elements are numbered from 0 upwards.

If VFP_WIDTH > FPR_WIDTH, each source or destination vector operand for these vector insert/extract/replicate operations must be an even-numbered floating-point register. Otherwise, an Illegal Instruction exception is raised.

Vector element 0 is always located in the low-order bits of the lowest-numbered floating-point register in each floating-point vector operand.

It is not illegal to perform replicate operations when VLEN(VFP_WIDTH, P) = 1.

Pseudo Code

```
                                        /* VFSREP*/
```

Assembly Code Example

```
VFSREP fd, fs1              ; replicate single-precision value across all
                            single-precision elements of a vector
```

Syntax and Encoding

		Instruction Code
VFSREP	fd, fs1	11100000010100101000EEEEE1LLLLL

VFSSQRT

Function

Single-precision floating-point square-root of all the elements in a vector

Extension Group

(HAS_FP == TRUE) && (FPU_DIV_OPTION == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = sqrt (s1)
```

Instruction Format

op fd, s1

Syntax Example

VFSSQRT fd, s1

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 signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 fd operand contains the square-root of all the single-precision elements of the s1 operand.

Pseudo Code

```
fd = sqrt (s1)                          VFSSQRT
```

Assembly Code Example

```
VFSSQRT fd, s1      fd operand contains the square-root of the s1 operand
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFSSQRT           fd, s1       11100000010000010100 0EEEEE 1LLLLL
```

VFSSUB

Function

Vector single-precision floating-point subtraction

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1-s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFSSUB fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 single-precision elements in a vector are subtracted from the corresponding single-precision elements in the second vector. The difference is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*VFSSUB
```

Assembly Code Example

```
VFSSUB fd, s1, s2    difference of s1 and s2 operands is stored in fd
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFSSUB            fd, s1, s2    11100mmm000100010 1MM0EEEEE1LLLLL
```

VFSSUBS

Function

Subtract a scalar from a vector with single-precision floating-point elements

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

```
fd = s1-s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFSSUBS fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are both +infinity or both are -infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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 scalar value is subtracted from each half-precision element in a vector. The difference is stored in the destination register.

Pseudo Code

```
fd = s1-s2                                  /*VFSSUBS
```

Assembly Code Example

```
VFSSUBS fd, s1, s2    ; vector floating-point subtraction of s1 and scalar s2
                      and the result is stored in fd.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

```
VFSSUBS            fd, s1, s2    11100mmm00010101 01MM0EEEEE1LLLLL
```

VFSSUBADD

Function

Instruction performs single-precision subtraction in all even-numbered lanes and addition in all odd-numbered lanes.

Extension Group

(HAS_FP == TRUE) && (FP_VEC_OPTION==TRUE)

Operation

On even-numbered lanes:

```
fd = s1 - s2
```

On odd-numbered elements:

```
fd = s1 + s2
```

Instruction Format

op fd, s1, s2

Syntax Example

VFSSUBADD fd, s1, s2

STATUS32 Flags Affected

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged

FPU_STATUS Flags

IV	•	Set when the operands are in the following combinations: s1= +infinity and s2 = -infinity or s2= -infinity and s2 = +infinity Or If any of the input operands is a signalling NaN when FP_CTRL.IVE==0 (invalid operation exception is disabled).
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

Instruction performs subtraction in all even-numbered lanes and addition in all odd-numbered lanes.

Pseudo Code

```
even-numbered lanes:                          /* VFSSUBADD */
  fd = s1 - s2
odd-numbered lanes:
  fd = s1 + s2
```

Assembly Code Example

```
VFSSUBADD fd, s1, s2    ;Instruction performs subtraction in all even-
                        numbered lanes and addition in all odd-numbered
                        lanes.
```

Syntax and Encoding

For more information about the encodings, see Table 8-7 and Table 8-8.

		Instruction Code
VFSSUBADD	fd, s1, s2	11100mmm00011111101MM0EEEEE1LLLLL

FSABS

Function

Single-precision floating-point absolute operation.

Alias

FSSGNJX f1, f2, f2

FHABS

Function

Half-precision floating-point absolute operation.

Alias

FHSGNJX f1, f2, f2

FDABS

Function

Double-precision floating-point absolute operation.

Alias

FDSGNJX f1, f2, f2

FSMOV

Function

Single-precision floating-point move operation.

Alias

FSSGNJ f1, f2, f2

FHMOV

Function

Half-precision floating-point move operation.

Alias

FHSGNJ f1, f2, f2

FSNEG

Function

Single-precision floating-point negate operation.

Alias

FSSGNJN f1, f2, f2

FHNEG

Function

Half-precision floating-point negate operation.

Alias

FHSGNJN f1, f2, f2

FDNEG

Function

Double-precision floating-point negate operation.

Alias

FDSGNJN f1, f2, f2

Part 5
Appendices

In this part:

- Implementation-dependent Behavior
- ARConnect Commands Details

A Implementation-dependent Behavior

A.1 EM Exception Handling

A.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.

A.2 Cause Codes for Memory Accesses and Data/Address Errors

The core-specific Parameter field may be defined by each core product to encode any information that is appropriate for that core, such as the type of memory accessed and whether it was an address or data error. ARC HS/EV/VPX5 processors fill these bits with zeros, whereas ARC EM inserts suitable values to encode the different types of CCM and data/address errors. See your processor Databook for more information on the core-specific parameter codes.

Cause of EV_MachineCheck Exception	Vector	Cause Code	Parameter
Internal Memory Error on Instruction Fetch	0x03	0x04	Core-specific
Internal Memory Error on Data Access	0x03	0x05	Core-specific
Illegal Overlapping MPU Entries	0x03	0x06	Core-specific
Secure vector table in normal memory	0x03	0x10	Core-specific
NSC jump table not in Secure memory	0x03	0x11	Core-specific

A.3 Exception Handling

A.3.1 ECR User mode Setting for the ARC HS6x/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.

A.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 A-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

B
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).

B.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.

B.1.1 Parameter Format

N/A

B.1.2 CONNECT_WDATA Register Format

N/A

B.1.3 CONNECT_READBACK Register Format

Figure B-1 CONNECT_READBACK Register Format for CMD_CHECK_CORE_ID Command

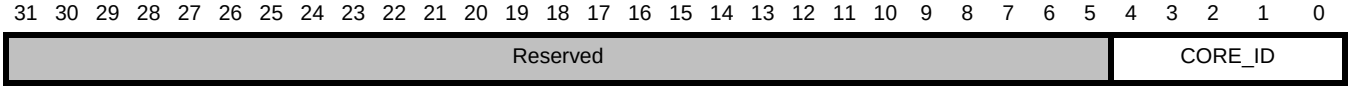


Table B-1 CONNECT_READBACK Register Field Descriptions for CMD_CHECK_CORE_ID

Field	Bits	Description
CORE_ID	[4:0]	Indicates core ID of the current core issuing CMD_CHECK_CORE_ID command 0 = CORE0 1 = CORE1 ..... 31 = CORE31

B.2 CMD_INTRPT_GENERATE_IRQ

Field	Value
CONNECT_CMD[7:0]	0x01

Each core can be an interrupt initiator core and use the `CMD_INTRPT_GENERATE_IRQ` command to generate an interrupt to another core or to external system in the system.

The `PARAMETER` field specified with this command is used to distinguish whether the interrupt is generated to a core or to the external system, and this field specifies the `CORE_ID` of the interrupt receiver core if the interrupt is generated to the core.

Each core is assigned an internal `INTRPT_STATUS` register in ICI to record the status of the interrupt (acknowledged or not) from it to all other cores. The `CMD_INTRPT_GENERATE_IRQ` command sets one bit in the interrupt initiator core's `INTRPT_STATUS` register corresponding to the interrupt receiver core.

The interrupt line of the interrupt receiver core is asserted after ARConnect issues this command and is asserted until the interrupt receiver core acknowledges all the inter-core interrupts it received.

B.2.1 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.

B.2.2 CONNECT_WDATA Register Format

N/A

B.2.3 CONNECT_READBACK Register Format

N/A

B.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 acknowledgment to the interrupt-initiating core.

If a level-sensitive interrupt is generated from a core to the external host, the host uses the `CMD_INTRPT_GENERATE_ACK` command through the peripheral slave port of the cluster network to provide the acknowledgment. (The mode of interrupt to external host can be programmed through the `CMD_INTPRT_EXT_MODE` command. By default, an interrupt to external host is pulse-sensitive).

The bit [8] of the PARAMETER field is used to specify whether the acknowledgment is from the core or from external host.

- 1: the acknowledge is from external host.
- 0: the acknowledge is from core.

The `PARAMETER` field is used to specify the `CORE_ID` of the interrupt initiator core.

The `CMD_INTRPT_GENERATE_ACK` command clears the bit that corresponds to the interrupt receiver core in the interrupt initiator core's `INTRPT_STATUS` register. However, the interrupt line of this interrupt receiver core is only de-asserted after it serves all the inter-core interrupts it has received.

If the acknowledgment is from the external host, the ICI component clears the `mcip_irq_<core_pfx>_to_host signal`.

B.3.1 Parameter Format

> **Note** If an interrupt to the external host is programmed as pulse-sensitive (through CMD_INTRPT_EXT_MODE), the command with parameter bit[8] as 1 is treated as NOP.

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.

B.3.2 CONNECT_WDATA Register Format

N/A

B.3.3 CONNECT_READBACK Register Format

N/A

B.4 CMD_INTRPT_READ_STATUS

Field	Value
CONNECT_CMD[7:0]	0x03

As an interrupt initiator, each core can use the CMD_INTRPT_READ_STATUS command to check the interrupt status generated from it to other cores.

The PARAMETER field is used to specify the CORE_ID of the interrupt receiver core. The CONNECT_READBACK register returns the value of the bit corresponding to the interrupt receiver core from the interrupt initiator's INTRPT_STATUS register.

B.4.1 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.

B.4.2 CONNECT_WDATA Register Format

N/A

B.4.3 CONNECT_READBACK Register Format

Figure B-2 CONNECT_READBACK Register Format for CMD_INTRPT_READ_STATUS Command

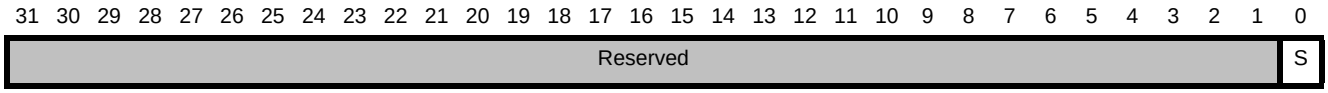


Table B-2 CONNECT_READBACK Register Field Descriptions for CMD_INTERRUPT_READ_STATUS

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

B.5 CMD_INTRPT_CHECK_SOURCE

Field	Value
CONNECT_CMD[7:0]	0x04

All the inter-core interrupts from other cores to the interrupt receiver core are combined into single interrupt line. An interrupt receiver core can use the CMD_INTRPT_CHECK_SOURCE command to find the source of an inter-core interrupt.

The CONNECT_READBACK register returns a vector gathered from the internal INTRPT_STATUS registers of all interrupt initiator cores.

B.5.1 Parameter Format

N/A

B.5.2 CONNECT_WDATA Register Format

N/A

B.5.3 CONNECT_READBACK Register Format

For an N-core system, the CONNECT_READBACK register returns a vector value after the core issues the CMD_INTRPT_CHECK_SOURCE command with each bit corresponding to a core. The following layout is for a 10-core system.

Figure B-3 CONNECT_READBACK Register Format for CMD_INTRPT_CHECK_SOURCE Command

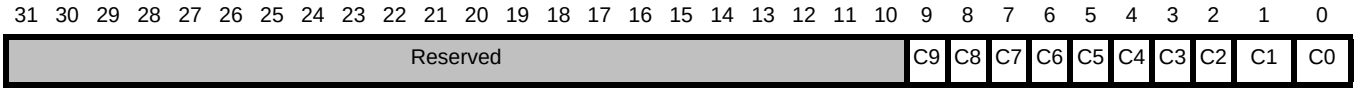


Table B-3 CONNECT_READBACK Register Field Descriptions for CMD_INTRPT_CHECK_SOURCE

Field	Bits	Description
C*n*	[n]	Indicates whether Core*n* is generating an interrupt to the current core. ■ `0` = Core*n* is not generating an interrupt to the current core. ■ `1` = Core*n* is generating an interrupt to the current core.

B.6 CMD_INTRPT_GENERATE_ACK_BIT_MASK

Field	Value
CONNECT_CMD[7:0]	0x05

Each core can be an interrupt receiver core and can use the CMD_INTRPT_GENERATE_ACK_BIT_MASK command to generate an interrupt acknowledgment to the interrupt-initiating core.

Note If an external master issues this command to ICI through ASI, the command is treated as a NOP.

The 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.

B.6.1 Parameter Format

N/A

B.6.2 CONNECT_WDATA Register Format

Each bit[n] is used to indicate 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

B.6.3 CONNECT_READBACK Register Format

N/A

B.7 CMD_INTRPT_EXT_MODE

Field	Value
CONNECT_CMD[7:0]	0x6

Each core or the external master can use the `CMD_INTRPT_EXT_MODE` to program the hardware behavior of interrupt signal to external host.

B.7.1 Parameter Format

The parameter[0] field specifies the interrupt hardware behavior mode:

- 1: generate level-sensitive interrupt to host.
- 0: generate pulse-sensitive interrupt to host.

B.7.2 CONNECT_WDATA Register Format

N/A

B.7.3 CONNECT_READBACK Register Format

N/A

B.8 CMD_INTRPT_SET_PULSE_CNT

Field	Value
CONNECT_CMD[7:0]	0x7

An internal counter `PULSE_PUCNT` is implemented in the ICI component to control the number of cycles asserted for a pulse-sensitive interrupt to external host. That is, if the `PULSE_CNT` is set as `5`, then the pulse sensitive interrupt lasts for 5 MCIP clock cycles. Each core or external master can use the `CMD_INTRPT_SET_PULST_CNT` command to set the `PULSE_CNT`.

The `CONNECT_WDATA` register is used to set the value of this pulse duration counter.

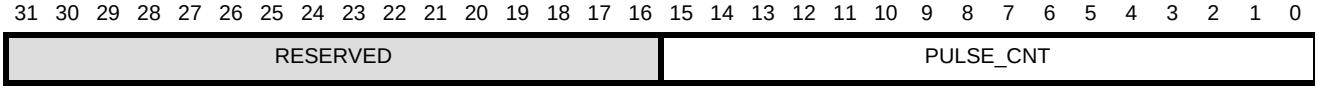
Note The pulse-sensitive interrupt must be of at least 2 MCIP cycles.

B.8.1 Parameter Format

N/As

B.8.2 CONNECT_WDATA Register Format

Figure B-4 CONNECT_READBACK Register Format for CMD_INTRPT_SET_PULSE_CNT Command



B.8.3 CONNECT_READBACK Register Format

N/A

B.9 CMD_INTRPT_READ_PULSE_CNT

Field	Value
CONNECT_CMD[7:0]	0x8

Each core or the external master can use the `CMD_INTRPT_READ_PULSE_CNT` to read the `PULSE_CNT` value.

The `CONNECT_READBACK` register is used to read the value of this pulse interrupt duration counter.

B.9.1 Parameter Format

N/A

B.9.2 CONNECT_WDATA Register Format

N/A

B.9.3 CONNECT_READBACK Register Format

The layout of the CONNECT_READBACK register is the same as the CONNECT_WDATA layout of the CMD_INTRPT_SET_PULSE_CNT command.

B.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.

B.10.1 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.

B.10.2 CONNECT_WDATA Register Format

N/A

B.10.3 CONNECT_READBACK Register Format

Figure B-5 CONNECT_READBACK Register Format for CMD_SEMA_CLAIM_AND_READ Command

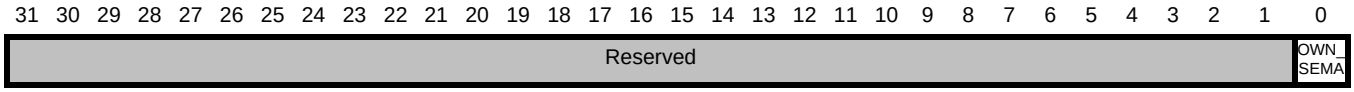


Table B-4 CONNECT_READBACK Register Field Descriptions for CMD_SEMA_CLAIM_AND_READ

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.

B.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.

B.11.1 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.

B.11.2 CONNECT_WDATA Register Format

N/A

B.11.3 CONNECT_READBACK Register Format

N/A

B.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.

B.12.1 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.

B.12.2 CONNECT_WDATA Register Format

N/A

B.12.3 CONNECT_READBACK Register Format

N/A

B.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.

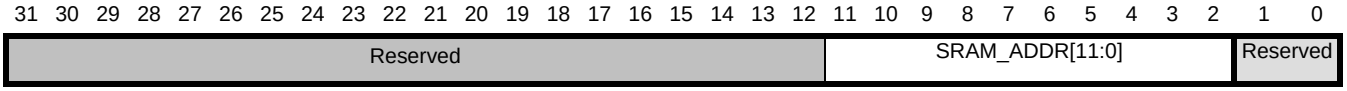
Each core is assigned an internal MSG_ADDR register in ICM to record the address of message-passing SRAM for it. The CONNECT_WDATA register contains the data to be written into the internal MSG_ADDR register.

B.13.1 Parameter Format

N/A

B.13.2 CONNECT_WDATA Register Format

Figure B-6 CONNECT_WDATA Register Format for CMD_MSG_SRAM_SET_ADDR Command



Note Read or write access to the message-passing SRAM is word-aligned, so the two least-significant bits of SRAM_ADDR are always ignored. And, the address exceeding the size of the SRAM field is truncated automatically.

B.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.

B.14.1 Parameter Format

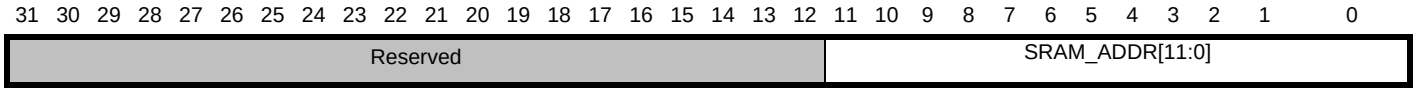
N/A

B.14.2 CONNECT_WDATA Register Format

N/A

B.14.3 CONNECT_READBACK Register Format

Figure B-7 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_ADDR Command



B.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.

B.15.1 Parameter Format

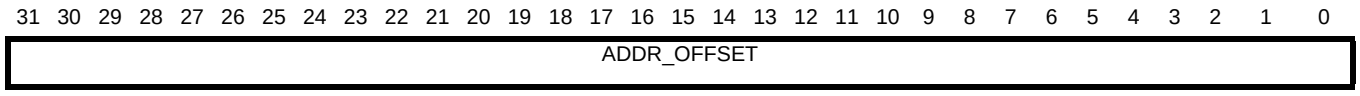
N/A

B.15.2 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 B-8.

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 B-8 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.

B.15.3 CONNECT_READBACK Register Format

N/A

B.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.

B.16.1 Parameter Format

N/A

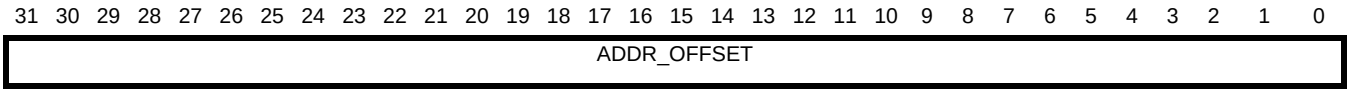
B.16.2 CONNECT_WDATA Register Format

N/A

B.16.3 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 B-9.

Figure B-9 CONNECT_READBACK Register Format for Read-Offset Command



B.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.

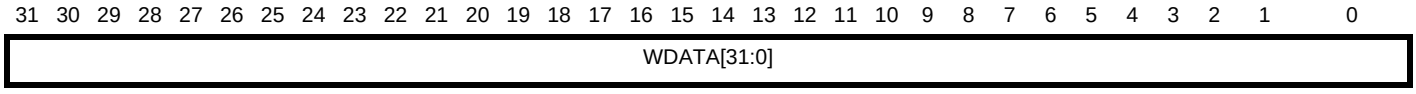
B.17.1 Parameter Format

N/A

B.17.2 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 B-10 CONNECT_WDATA Register Format for CMD_MSG_SRAM_WRITE Command



B.17.3 CONNECT_READBACK Register Format

N/A

B.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.

B.18.1 Parameter Format

N/A

B.18.2 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 B-10.

B.18.3 CONNECT_READBACK Register Format

N/A

B.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.

Note The `CMD_MSG_SRAM_WRITE_IMM` command does not update the value of core's `MSG_ADDR` register.

B.19.1 Parameter Format

The `Parameter[11:0]` field specifies the SRAM address.

B.19.2 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 B-10.

B.19.3 CONNECT_READBACK Register Format

N/A

B.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.

B.20.1 Parameter Format

N/A

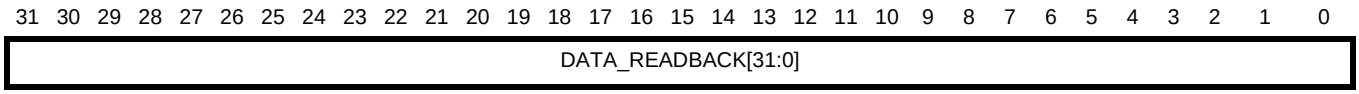
B.20.2 CONNECT_WDATA Register Format

N/A

B.20.3 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 B-11 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ Command



B.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.

B.21.1 Parameter Format

N/A

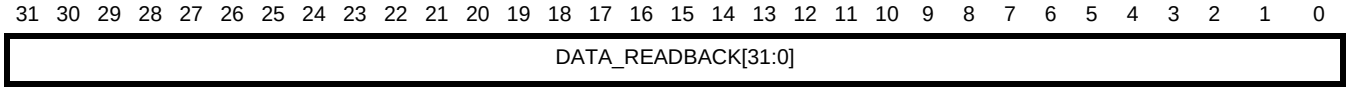
B.21.2 CONNECT_WDATA Register Format

N/A

B.21.3 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 B-12 CONNECT_READBACK Register Format for CMD_MSG_SRAM_READ_INC Command



B.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.

B.22.1 Parameter Format

The Parameter [11:0] field specifies the SRAM address by an immediate.

B.22.2 CONNECT_WDATA Register Format

N/A

B.22.3 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.

B.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 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.

B.23.1 Parameter Format

N/A

B.23.2 CONNECT_WDATA Register Format

The data must be written to the CONNECT_WDATA register before issuing the CMD_MSG_SET_ECC_CTRL command.

Figure B-13 CONNECT_WDATA Register Format for CMD_MSG_SET_ECC_CTRL Command

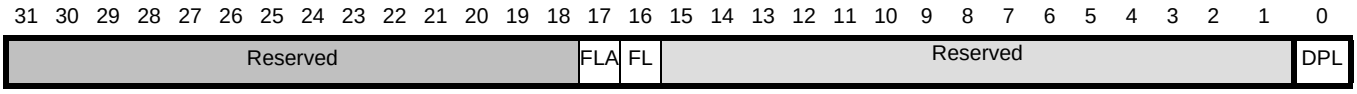


Table B-5 CONNECT_WDATA Register Field Description for CMD_MSG_SET_ECC_CTRL

Field	Bits	Description
DPL	[0]	Control bit for ECC protection on the Message SRAM; ■ 0 = enable protection ■ 1 = disable protection On reset, the control bits are set to 0; that is ECC protection is enabled. NOTE: If protection is disabled, the FL flag is never set. When ECC 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: 2-bits errors in data or address (in SECDED configuration) and 2-bit error in address NOTE: The FL bit can only be set by the ECC hardware logic; it cannot be set by software. But software can use the CMD_MSG_SET_ECC_CRTL command to write 0 to clear this bit. And software can also use the CMD_MSG_READ_ECC_CTRL to read the FL bit. If the software writes 0 and ECC hardware logic also sets this bit simultaneously, the software will preempt.
FLA	[17]	Address single-bit error flag. The FLA bit can only be asserted by the ECC hardware logic; it cannot be asserted by software. software can use the MSG_SET_ECC_CRTL command to write 0 to clear this bit, and use the CMD_MSG_READ_ECC_CTRL command to read the FLA value. ■ 0: No 1-bit address ECC error detected. This is the default value ■ 1: 1-bit address ECC error. This bit is asserted when 1-bit address error is detected. If the software writes 0 and ECC hardware logic also asserts this bit simultaneously, the software preempts. If address MSGECCA or MSGECC is disabled, write this bit is ignored and read this bit returns 0.

B.23.3 CONNECT_READBACK Register Format

N/A.

B.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.

B.24.1 Parameter Format

N/A

B.24.2 CONNECT_WDATA Register Format

N/A

B.24.3 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.

B.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.

B.25.1 Parameter Format

N/A

B.25.2 CONNECT_WDATA Register Format

N/A

B.25.3 CONNECT_READBACK Register Format

The `CONNECT_READBACK` returns the value of the internal `ICM_SBE_CNT` register as shown in Figure B-14.

Figure B-14 CONNECT_READBACK Register Format for CMD_MSG_READ_ECC_SBE_CNT Command

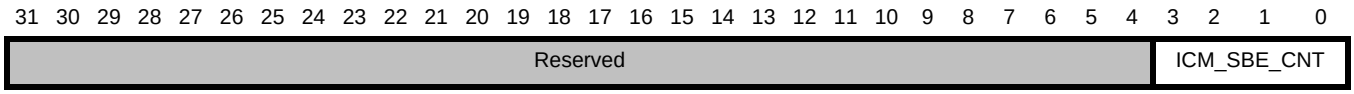


Table B-6 CONNECT_READBACK Register Field Description for CMD_MSG_READ_ECC_SBE_CNT

Field	Bits	Description
SBE_CNT	[3:0]	ECC error counter. It records the numbers of the 1-bit ECC error correction.

B.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.

B.26.1 Parameter Format

N/A

B.26.2 CONNECT_WDATA Register Format

N/A

B.26.3 CONNECT_READBACK Register Format

N/A

B.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.

B.27.1 Parameter Format

N/A

B.27.2 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.

Each bit in the MCD_CORE register identifies a core. For an N-core system, this register has N bits. For a 10-core system, the layout of the CONNECT_WDATA register is shown in Figure B-15.

Figure B-15 CONNECT_WDATA Register Format for CMD_DEBUG_RESET Command

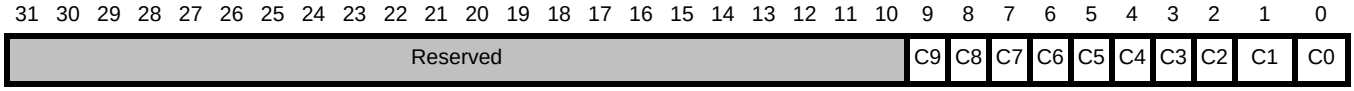


Table B-7 CONNECT_WDATA Field Descriptions for CMD_DEBUG_RESET

Field	Bits	Description
Cn	n	Coren select. Indicates whether coren is selected for the current command ■ 0 = Not selected ■ 1 = Coren is selected

B.27.3 CONNECT_READBACK Register Format

N/A

B.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.

B.28.1 Parameter Format

N/A

B.28.2 CONNECT_WDATA Register Format

The layout of CONNECT_WDATA register is same as of CMD_DEBUG_RESET command.

B.28.3 CONNECT_READBACK Register Format

N/A

B.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.

B.29.1 Parameter Format

N/A

B.29.2 CONNECT_WDATA Register Format

The layout of CONNECT_WDATA register is same as of CMD_DEBUG_RESET command.

B.29.3 CONNECT_READBACK Register Format

N/A

B.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.

B.30.1 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

B.30.2 CONNECT_WDATA Register Format

The layout of CONNECT_WDATA register is same as of CMD_DEBUG_RESET command.

B.30.3 CONNECT_READBACK Register Format

N/A

B.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.

B.31.1 Parameter Format

N/A

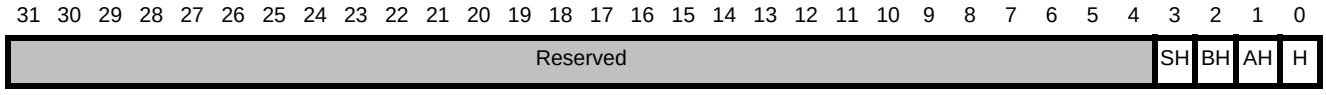
B.31.2 CONNECT_WDATA Register Format

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.

B.31.3 CONNECT_READBACK Register Format

Figure B-16 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).

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.

B.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 SELECT register in ICD is used to determine the cores in the system that should be conditionally 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 SELECT register.

B.32.1 Parameter Format

N/A

B.32.2 CONNECT_WDATA Register Format

The layout of CONNECT_WDATA register is same as of CMD_DEBUG_RESET command.

B.32.3 CONNECT_READBACK Register Format

N/A

B.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

B.33.1 Parameter Format

N/A

B.33.2 CONNECT_WDATA Register Format

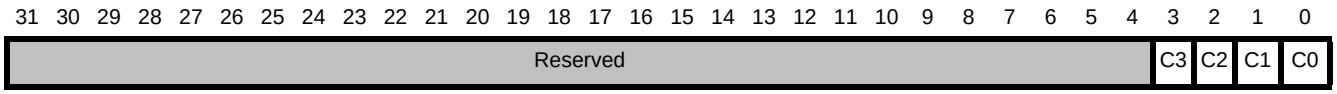
N/A

B.33.3 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. The number of valid bits in this register equals the number of cores.

For a four-core configuration as the example, the register layout is shown in Figure B-17.

Figure B-17 CONNECT_READBACK Register Format for CMD_DEBUG_READ_SELECT Command



B.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.

B.34.1 Parameter Format

N/A

B.34.2 CONNECT_WDATA Register Format

N/A

B.34.3 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 B-18.

Figure B-18 CONNECT_READBACK Register Format for CMD_DEBUG_READ_EN

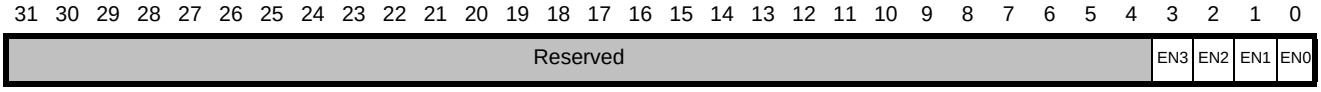


Table B-8 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_EN

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

B.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.

B.35.1 Parameter Format

N/A

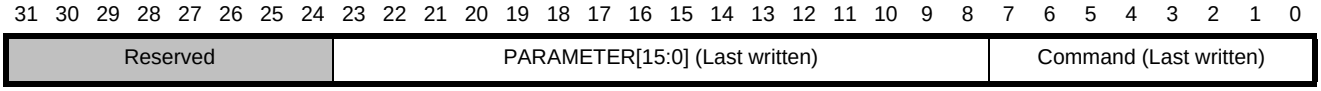
B.35.2 CONNECT_WDATA Register Format

N/A

B.35.3 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 B-19.

Figure B-19 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CMD Command



B.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.

B.36.1 Parameter Format

N/A

B.36.2 CONNECT_WDATA Register Format

N/A

B.36.3 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 B-20.

Figure B-20 CONNECT_READBACK Register Format for CMD_DEBUG_READ_CORE Command

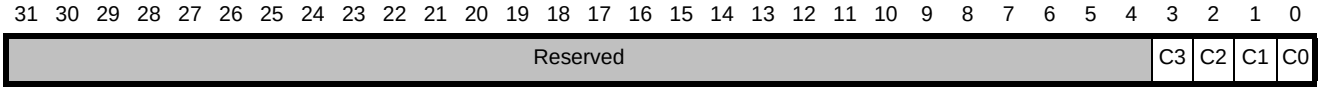


Table B-9 CONNECT_READBACK Register Field Descriptions for CMD_DEBUG_READ_CORE

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

B.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.

B.37.1 Parameter Format

N/A

B.37.2 CONNECT_WDATA Register Format

N/A

B.37.3 CONNECT_READBACK Register Format

N/A

B.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.

B.38.1 Parameter Format

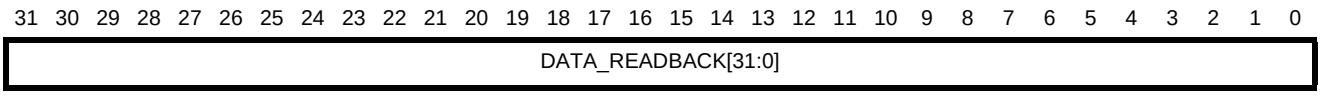
N/A

B.38.2 CONNECT_WDATA Register Format

N/A

B.38.3 CONNECT_READBACK Register Format

Figure B-21 CONNECT_READBACK Register Format for CMD_GFRC_READ_LO Command



B.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.

B.39.1 Parameter Format

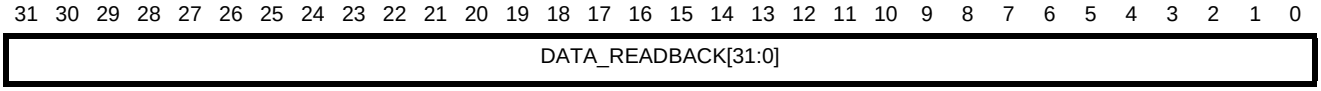
N/A

B.39.2 CONNECT_WDATA Register Format

N/A

B.39.3 CONNECT_READBACK Register Format

Figure B-22 CONNECT_READBACK Register Format for CMD_GFRC_READ_HI Command



B.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.

B.40.1 Parameter Format

N/A

B.40.2 CONNECT_WDATA Register Format

N/A

B.40.3 CONNECT_READBACK Register Format

N/A

B.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.

B.41.1 Parameter Format

N/A

B.41.2 CONNECT_WDATA Register Format

N/A

B.41.3 CONNECT_READBACK Register Format

N/A

B.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.

B.42.1 Parameter Format

N/A

B.42.2 CONNECT_WDATA Register Format

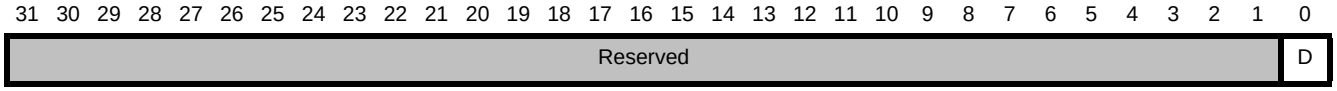
N/A

B.42.3 CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the content of internal DISABLE register after CMD_GFRC_READ_DISABLE command.

Figure B-23 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command.

Figure B-23 CONNECT_READBACK Register Format for CMD_GFRC_READ_DISABLE Command



B.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.

B.43.1 Parameter Format

N/A

B.43.2 CONNECT_WDATA Register Format

Figure B-24 shows CONNECT_WDATA Register Format for Commands Updating the internal HALT register.

Figure B-24 CONNECT_WDATA Register Format for Command CMD_GFRC_SET_CORE

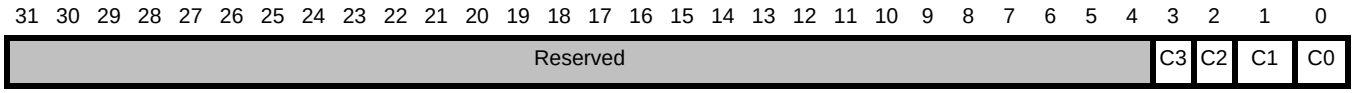


Table B-10 CONNECT_WDATA Field Descriptions

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

Table B-10 CONNECT_WDATA Field Descriptions

Field	Bits	Description
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

B.43.3 CONNECT_READBACK Register Format

N/A

B.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.

B.44.1 Parameter Format

N/A

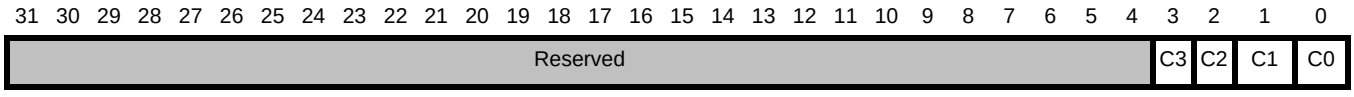
B.44.2 CONNECT_WDATA Register Format

N/A

B.44.3 CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal CORE register.

Figure B-25 CONNECT_READBACK Register Format for Command CMD_GFRC_READ_CORE



B.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.

B.45.1 Parameter Format

N/A

B.45.2 CONNECT_WDATA Register Format

N/A

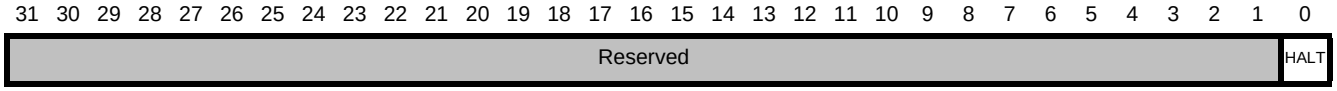
B.45.3 CONNECT_READBACK Register Format

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 B-26 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command.

Figure B-26 CONNECT_READBACK Register Format for CMD_GFRC_READ_HALT Command



B.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.

B.46.1 Parameter Format

N/A

B.46.2 CONNECT_WDATA Register Format

N/A

B.46.3 CONNECT_READBACK Register Format

N/A

B.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.

B.47.1 Parameter Format

N/A

B.47.2 CONNECT_WDATA Register Format

N/A

B.47.3 CONNECT_READBACK Register Format

N/A

B.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.

B.48.1 Parameter Format

N/A

B.48.2 CONNECT_WDATA Register Format

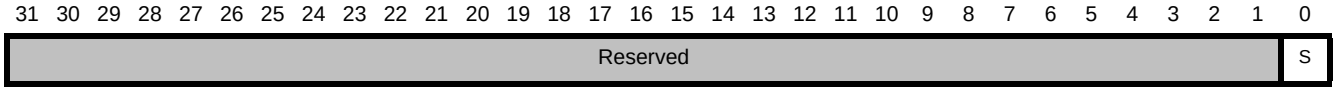
N/A

B.48.3 CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the content of internal clock status register after CMD_GFRC_READ_CLK_STATUS command.

Figure B-27 shows CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command.

Figure B-27 CONNECT_READBACK Register Format for CMD_GFRC_READ_CLK_STATUS Command



When read result value is 0, it indicates that current GFRC clock is running.

When read result value is 1, it indicates that current GFRC clock is gated off.

B.49 CMD_GFRC_READ_FULL

Field	Value
CONNECT_CMD[7:0]	0x4D

Each core that supports 64-bit auxiliary access or external master can use CMD_GFRC_READ_FULL command to read the 64-bits of the GFRC counter.

The CONNECT_READBACK_64 register returns the 64-bit data.

B.49.1 Parameter Format

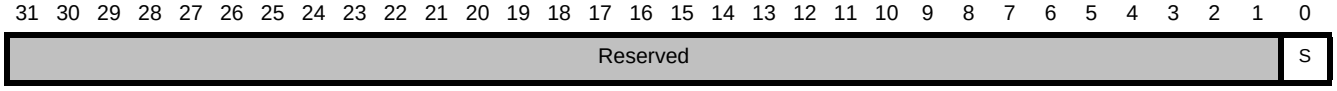
N/A

B.49.2 CONNECT_WDATA Register Format

N/A

B.49.3 CONNECT_READBACK_64 Register Format

Figure B-28 CONNECT_READBACK Register Format for CMD_GFRC_READ_FULL Command



B.50 CMD_IDU_ENABLE

Field	Value
CONNECT_CMD[7:0]	0x71

IDU has a global `ENABLE` register to disable or enable the common interrupts. Each core or the external master can use the `CMD_IDU_ENABLE` command to enable the common interrupts.

The global `ENABLE` register is zero after reset.

B.50.1 Parameter Format

N/A

B.50.2 CONNECT_WDATA Register Format

N/A

B.50.3 CONNECT_READBACK Register Format

N/A

B.51 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.

B.51.1 Parameter Format

N/A

B.51.2 CONNECT_WDATA Register Format

N/A

B.51.3 CONNECT_READBACK Register Format

N/A

B.52 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.

B.52.1 Parameter Format

N/A

B.52.2 CONNECT_WDATA Register Format

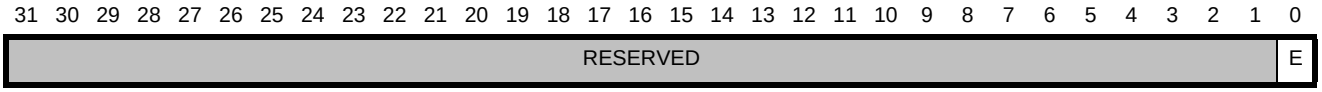
N/A

B.52.3 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 B-29. A value of 1 indicates that IDU is enabled, otherwise it is disabled.

Figure B-29 CONNECT_READBACK Register Format for CMD_IDU_READ_ENABLECommand



B.53 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 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.

B.53.1 Parameter Format

The PARAMETER[8] field identifies the destination core with index in the range 0 to 31 or in the range 32 to 63. This parameter field is ignored if the total number of cores is not greater than 32.

- 1: the destination core index is in the range 32 ~ 63.

0: the destination core index is in the range 0 ~ 31.The Parameter [6:0] field defines the index of a common interrupt, for example 7'd0 indicates the common interrupt 0; 7'd8 indicates the common interrupt 8.

B.53.2 CONNECT_WDATA Register Format

Figure B-30 CONNECT_WDATA Register Format for CMD_IDU_SET_MODE Command

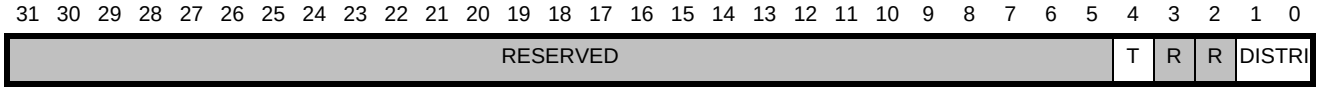


Table B-11 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MODE

Field	Bits	Description
DISTRI	[1:0]	The distribution mode: ■ 0x0: Round-robin mode ■ 0x1: First-acknowledge mode ■ 0x2: All-destination mode

Table B-11 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MODE

Field	Bits	Description
T	[4]	The trigger mode: ■ 0: level triggered ■ 1: edge triggered NOTE: This field is invalid for common interrupts programmed by the software.

B.53.3 CONNECT_READBACK Register Format

N/A

B.54 CMD_IDU_READ_MODE

Field	Value
CONNECT_CMD[7:0]	0x75

Each core can use the CMD_IDU_READ_MODE command to read the internal MODE 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.

B.54.1 Parameter Format

The Parameter [6:0] field defines the index of a common interrupt.

B.54.2 CONNECT_WDATA Register Format

N/A

B.54.3 CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the contents of the internal MODE register of the specified common interrupt after the CMD_IDU_READ_MODE command, the layout of it is same as the CONNECT_WDATA register of the CMD_IDU_SET_MODE command.

B.55 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 to receive the specified common interrupt when it is triggered.

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.

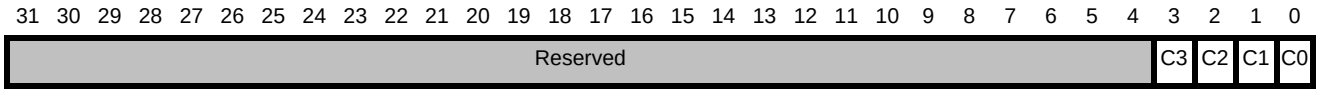
B.55.1 Parameter Format

The Parameter [6:0] field defines the index of a common interrupt.

B.55.2 CONNECT_WDATA Register Format

For a four-core system, the layout of the CONNECT_WDATA register is shown in Figure B-31.

Figure B-31 CONNECT_WDATA Register Format for CMD_IDU_SET_DEST Command



The field descriptions are shown in Table B-12.

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 B-12 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
C3	[3]	Indicates whether Core3 is targeted for the specified common interrupt ■ 0 = Not targeted ■ 1 = Core3 is targeted

B.55.3 CONNECT_READBACK Register Format

N/A

B.56 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.

B.56.1 Parameter Format

The Parameter [6:0] field defines the index of the common interrupt.

B.56.2 CONNECT_WDATA Register Format

N/A

B.56.3 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.

B.57 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.

B.57.1 Parameter Format

The Parameter [6:0] field specifies the index of the common interrupt.

B.57.2 CONNECT_WDATA Register Format

N/A

B.57.3 CONNECT_READBACK Register Format

N/A

B.58 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.

B.58.1 Parameter Format

The `Parameter [6:0]` field specifies the index of the common interrupt.

B.58.2 CONNECT_WDATA Register Format

N/A

B.58.3 CONNECT_READBACK Register Format

N/A

B.59 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.

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.

B.59.1 Parameter Format

The Parameter [6:0] field defines the index of the common interrupt.

B.59.2 CONNECT_WDATA Register Format

N/A

B.59.3 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 B-32.

Figure B-32 CONNECT_READBACK Register Format for CMD_IDU_CHECK_STATUS Command

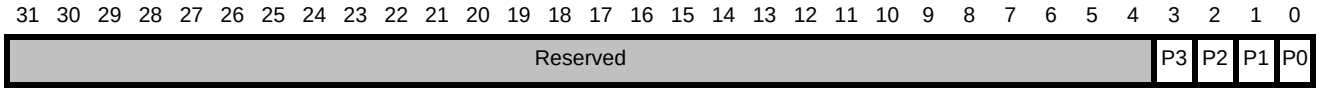


Table B-13 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_STATUS

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

Table B-13 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_STATUS

Field	Bits	Description
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

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.

B.60 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.

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.

B.60.1 Parameter Format

The Parameter [6:0] field defines the index of the common interrupt.

B.60.2 CONNECT_WDATA Register Format

N/A

B.60.3 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 B-33.

Figure B-33 CONNECT_READBACK Register Format for CMD_IDU_CHECK_SOURCE command

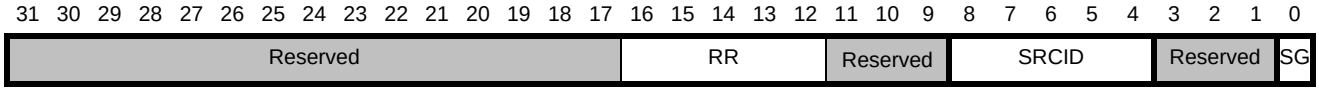


Table B-14 CONNECT_READBACK Register Field Description for CMD_IDU_CHECK_SOURCE

Field	Bits	Description
SG	[0]	**Software generation flag bit** ■ 0 = the interrupt is not generated by software by using the CMD_IDU_GEN_CIRQ command. ■ 1 = the interrupt is generated by software by using the CMD_IDU_GEN_CIRQ command. NOTE: The SG bit is cleared by the CMD_IDU_ACK_CIRQ 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 CORE_ID 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. The external master is indicated by the last core ID +1 (so it is 0x2 for a 2 core system and 0x4 for a 4 core system). When the SG bit is 1, the SRCID can be overridden with another CORE_ID if that core sends the CMD_IDU_GEN_CIRQ 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.

B.61 CMD_IDU_SET_MASK

Field	Value
CONNECT_CMD[7:0]	0x7C

Each common interrupt is assigned an internal MASK register in the IDU. Each core can use the CMD_IDU_SET_MASK command to mask or unmask the specified common interrupt.

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.

B.61.1 Parameter Format

The Parameter [6:0] field defines the index of a common interrupt.

B.61.2 CONNECT_WDATA Register Format

Figure B-34 CONNECT_WDATA Register Format for CMD_IDU_SET_MASK command

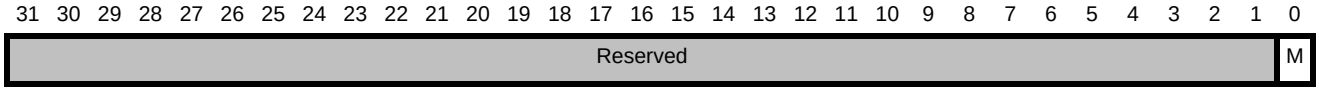


Table B-15 CONNECT_WDATA Register Field Description for CMD_IDU_SET_MASK

Field	Bits	Description
M	[0]	■ 0: Unmask this interrupt ■ 1: Mask this interrupt

B.61.3 CONNECT_READBACK Register Format

N/A

B.62 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.

B.62.1 Parameter Format

The Parameter [6:0] field defines the index of the common interrupt.

B.62.2 CONNECT_WDATA Register Format

N/A

B.62.3 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.

B.63 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.

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.

B.63.1 Parameter Format

The Parameter [6:0] field defines the index of a common interrupt.

B.63.2 CONNECT_WDATA Register Format

N/A

B.63.3 CONNECT_READBACK Register Format

The CONNECT_READBACK register returns the value to indicate if it is the first-acknowledging core, as shown in Figure B-35.

Figure B-35 CONNECT_READBACK Register Format for CMD_IDU_CHECK_FIRST Command

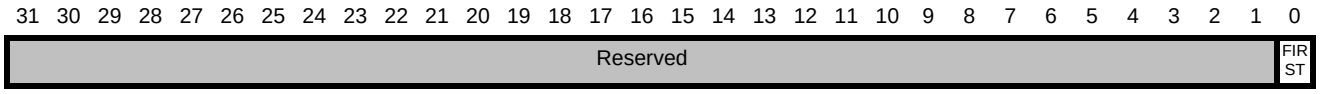


Table B-16 CONNECT_WDATA Register Field Description for CMD_IDU_CHECK_FIRST

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.

B.64 Examples: ARConnect ICI

B.64.1 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 B-1.

Example B-1 Segment Program of Scenario 1 — Core1

`chk status:`	Check the interrupt status of Core2, so that any previous interrupt is not overridden.
`SR 0x00000203,[CONNECT_CMD]`	Send the `CMD_INTRPT_READ_STATUS` command to check the status of the interrupt it generated to Core2.
`LR r1, [CONNECT_READBACK]` `CMP r1, 1`	The interrupt status is read back using the `CONNECT_READBACK` register.
`BEQ chk_status` … …	Exit the loop until the corresponding status bit is cleared.
`SR 0x00000201, [CONNECT_CMD]`	Send the `CMD_INTRPT_GENERATE_IRQ` command to generate inter-core interrupt to Core2.

Core2 is the interrupt receiver core. A program segment of its interrupt handler is shown in Example B-2.

Example B-2 Segment Program of Scenario 1 — Core2

`SR r2, [CONNECT_CMD]`	The content of r2 is `0x00000004`, indicating that the `CMD_INTRPT_CHECK_SOURCE` command is being sent to check the source of the interrupt.
`LR r4, [CONNECT_READBACK]`	Read back the vector value from the `CONNECT_READBACK` register to core register r4.
`AND.F 0, r4, 0x00000002` `BEQ gen_ack_to_c1`	Check whether the bit corresponding to Core1 is asserted.

`gen_ack_to_c1:` `SR r5, [CONNECT_CMD]`	The content of r5 is `0x00000102`, indicating that the `CMD_INTRPT_GENERATE_ACK` command is issued to acknowledge the interrupt from Core1. Note: Make sure the interrupt has been de-asserted before exiting the interrupt handler, for example, by using `CMD_INTRPT_CHECK_SOURCE` command again in a loop until the initiator core has de-asserted the interrupt.

B.64.2 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 B-3 Segment Program of Scenario 2 - Core0, Core1, Core2

`chk_status:`	Check the interrupt status of Core3, so that any previous interrupt is not overridden.
`SR 0x00000303, [CONNECT_CMD]`	Send the `CMD_INTRPT_READ_STATUS` command.
`LR r1, [CONNECT_READBACK]` `CMP.F r1, 0x00000001`	The interrupt status is read back from the `CONNECT_READBACK` register.
`BEQ chk_status`	Exit the loop until the corresponding status bit is cleared.
`SR 0x00000301, [CONNECT_CMD]`	Send the `CMD_INTRPT_GENERATE_IRQ` command to Core3 to generate inter-core interrupt.

…

Core3 is the interrupt receiver; a segment program of its interrupt handler is shown in Example B-4.

Example B-4 Segment Program of Scenario 2 - Core3

`serve_irq:`	
`SR r0, [CONNECT_CMD]`	The content of r0 is `0x00000040`, which indicates that the `CMD_INTRPT_CHECK_SOURCE` command is being sent to check the sources of the generated interrupts.
`LR r1,[CONNECT_READBACK]`	The vector value is read back from the `CONNECT_READBACK` register to core register r1.
`CMP r1, 0`	Check whether all interrupts have been serviced.

```
    BEQ    end_serve_irq
   ...
```

```
   MOV r2, 1
```

Set the start value 1 for index control.

```
serve_irq_for_each_core:
```

```
    MOV r3, r1
```

Move the interrupt source vector to temporary register r3.

```
    SUB   r10, r2,1
```

Generate the `shift_loop` control value.

```
shift_loop:
```

```
    CMP r10, 0
```

```
    BEQ gen_ack_to_each_core
```

If equal, go to generate interrupt acknowledges.

```
    LSR  r3, r3
```

Shift right operation of interrupt source vector.

```
    SUB  r10, r10,1
   BAL  shift_loop
```

```
gen_ack_to_each_core:
```

```
   AND.F 0, r3, 0x00000001
```

Check the value of bit0 for shifted interrupt source vector.

```
   BEQ       incr_index
```

If the AND result is zero, do nothing and increment the index; otherwise, go on, generate interrupt, and acknowledge to the core with a specific index value.

```
…
     SR    r11, [CONNECT_CMD]
…
```

Instructions to generate the command value and store it into r11. In this case, the content of r11 is sent using the command `CMD_INTRPT_GENERATE_ACK` to Core0, Core1, and Core2 respectively.

```
incr_index:
```

```
   ADD r2, r2, 1
```

```
   CMP r2, CONNECT_CORE_NUMBER
```

```
    BLS serve_irq_for_each_core
```

If the index value is less than or equal to the number of cores in the system, go on to check the interrupt status.

```
end_serve_irq:
```

```
   RTIE
```

Note: Ensure that the corresponding bit of the interrupt source has been cleared before exiting the handler.

B.65 Examples: ARConnect Semaphore Unit

B.65.1 Scenario 1

In this scenario, Core1 claims the ownership of semaphore 8. After some processing, Core1 releases semaphore 8.

Example B-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.

B.65.2 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 B-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 …	

`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).

B.66 Examples: ICD

B.66.1 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 B-7 shows a program segment of Core2 if the debugger programs ICD through Core2.

Example B-7 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.

B.66.2 Selectively Halt Cores Scenario

Core0 programs ICD to selectively halt other cores in the system.

Example B-8 shows a program segment of Core0.

Example B-8 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.

B.66.3 Selectively Run Cores Scenario

All cores have already halted. The debugger programs ICD to run all cores in the system.

Example B-9 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.

B.66.4 Selectively Reset Cores Scenario

Core0 is the master CPU of a two core system.

Example B-10 shows a program segment of Core0.

Example B-10 Selectively Reset Cores Scenario

wait_core1_halt:

`SR 0x00000038, [CONNECT_CMD]` `LR r0, [CONNECT_READBACK]`	Send the `CMD_DEBUG_READ_EN` command to ICD.
`AND.F 0, r0, 0x00000002` `BNZ wait_core1_halt`	Check the `EN` bit of Core1. 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 B-11 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.
…	

B.67 Examples: GFRC

Example B-12 Segment Program to Read Value of GFRC

`SR 0x00000042, [CONNECT_CMD]`	Send the `CMD_GRTC_READ_LO` command.
`LR r5, [CONNECT_READBACK]`	The lower 32-bit value of GRTC are returned to the `CONNECT_READBACK` register, and stored to r5 register of the core.
`SR 0x00000043, [CONNECT_CMD]`	Send the `CMD_GRTC_READ_HI command.`
`LR r6, [CONNECT_READBACK]`	The higher 32-bit value of GRTC is returned to the `CONNECT_READBACK` register, and store to r6 register of the core.

B.68 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 B-13 shows a segment program of the initiator core - Core0.

Example B-13 Segment Program of Common Interrupt 1 Generated by Core 0

```
_reset:                                  Reset the exception base
  .long  _start
  ...

irq25: .long irq_idu_handler             Common interrupt1 corresponds to IRQ25

test_start:                              On-reset start point of the program

.text
.global _start
_start:
  ...
  SR 0x00000072, [CONNECT_CMD]           Use the CMD_IDU_DISABLE command to disable the interrupt
                                         distribution unit
  SR 0x00000011, [CONNECT_WDATA]         Set the distribution mode of common interrupt 1.

  SR 0x00000174, [CONNECT_CMD]           Send the CMD_IDU_SET_MODE command to ARConnect,
                                         parameter [6:0] = 7'd01 indicates common interrupt1 is
                                         set
  SR 0x0000000f, [CONNECT_WDATA]         Four cores are targeted
  SR 0x00000176, [CONNECT_CMD]           Send the CMD_IDU_SET_DEST command to ARConnect,
                                         parameter [6:0] = 7'd01 indicates that the data in
                                         CONNECT_WDATA is written into the DEST register of common
                                         interrupt1
  SR 0x00000000, [CONNECT_WDATA]         Set the common interrupt MASK to 0
  SR 0x0000017C, [CONNECT_CMD]           Send the CMD_IDU_SET_MASK command to ARConnect,
                                         parameter [6:0] = 7'd01 to indicate that the common
                                         interrrrupt1 will be unmasked.
  SR 0x00000071, [CONNECT_CMD]           Send the CMD_IDU_ENABLE command to enable the interrupt
                                         distribution unit
  SR 0x00000178, [CONNECT_CMD]           Send the CMD_IDU_GEN_CIRQ command to ARConnect,
                                         parameter [6:0] = 7'd0 to indicate that the common
                                         interrupt1 is asserted.
```